Resource Centered Store

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Dipl.-Inf. Ralf Heese

Gutachter:

1. Prof. Johann-Christoph Freytag, Ph.D.
2. Prof. Dr. Felix Naumann
3. Prof. Dr. Martin Theobald

eingereicht am: 30. März 2015
Tag der mündlichen Prüfung: 28. September 2015

Man kennt nur die Dinge, die man zähmt.

Aus „Der kleine Prinz"
Antoine de Saint-Exupéry

Abstract

The Resource Description Framework (RDF) is the conceptual foundation for representing properties of real-world or virtual objects, so-called resources, and for describing the relationships between them. Standards based on RDF allow machines to access and process information automatically as well as to locate additional information about resources. Furthermore, it supports machines to relate resources to each other in a meaningful way.

The smallest information unit in RDF are triples which form a directed labeled multi-graph. As a consequence of the graph based model the corresponding query language SPARQL is also based on a graph model. Research results showed that relational and object-oriented database management systems (DBMS) have difficulties to store and query RDF data efficiently.

The most performant DBMS for managing and querying RDF data implement a RDF specific (native) storage model which is based on a set of B+ tree indexes. The RDF graph is split up into triples and stored in the indexes. The key disadvantages of these systems are the increased usage of secondary storage in cause of redundantly stored triples as well as the necessity of expensive join operation to compute the solutions of a SPARQL query.

In this work we develop and describe the Resource Centered Store which exploits RDF inherent characteristics to avoid the requirement for storing triples redundantly while improving the query performance of larger queries. In the RCS storage model triples are grouped by their first component (subject) and storing these star-shaped subgraphs on database pages – similar to relational DBMS. As a result the RCS can benefit from principles and algorithms that have been developed in the context of relational databases.

Additionally, we defined transformation rules and heuristics to optimize SPARQL queries and generate an efficient query execution plan. In this context we also defined graph pattern based indexes and investigated their benefits for computing the solutions of queries.

We implemented the RCS storage model prototypically and compared it to the native RDF DBMS Jena TDB. Our experiments showed that our storage model is especially suited to speed up the query performance of large star-shaped graph pattern.

Zusammenfassung

Mit dem Resource Description Framework (RDF) wurde eine konzeptionelle Grundlage geschaffen, um Eigenschaften von und die Beziehungen zwischen realen und virtuellen Objekten – so genannten Ressourcen – maschinenverarbeitbar zu beschreiben. Dadurch werden diese Daten für Maschinen zugänglicher und können unter anderem automatisch Daten zu einer Ressource lokalisieren und verarbeiten, unterschiedliche Bedeutungen einer Zeichenkette erkennen und implizite Informationen ableiten.

Das Datenmodell von RDF basiert auf gerichteten und beschrifteten Multigraphen, wobei die kleinste Informationseinheit aus einem Tripel besteht. Die zugehörige Anfragesprache SPARQL fundiert ebenfalls auf einem Graphmodell. Forschungsergebnisse haben gezeigt, dass relationale und objektorientierte Datenbankmanagementsysteme (DBMS) zum Verwalten von RDF-Daten ungeeignet sind.

Die leistungsstärksten DBMS für das Speichern und Anfragen realisieren daher ein RDF-spezifisches (natives) Speichermodell, das die RDF-Daten in einer Menge von B+-Bäumen ablegen. Der Leistungsgewinn dieser Systeme bedingt einen höheren Speicherbedarf durch redundantes Speichern von Tripeln in mehreren B+-Bäumen. Insbesondere sind auch bei diesen Systemen Join-ähnliche Operationen zum Berechnen der Lösungen einer Anfrage erforderlich, was bei größeren Anfragen zu Leistungseinbußen führt.

In dieser Arbeit wird der Resource Centered Store (RCS) entwickelt, dessen Speichermodell RDF-inhärente Eigenschaften ausnutzt, um Anfragen ohne die Notwendigkeit redundanter Speicherung effizient beantworten zu können. Die grundlegende Idee des RCS-Speichermodells besteht im Gruppieren der Tripel anhand ihrer ersten Komponente (Subjekt) und der Verwaltung von diesen sternförmigen Teilgraphen auf Datenbankseiten, ähnlich wie in relationalen DBMS. Dies führt unter anderem auch dazu, dass Prinzipien und Algorithmen von relationalen DBMS zur Beantwortung von Anfragen wiederverwendet werden können.

Darüber hinaus werden in dieser Arbeit Transformationsregeln und Heuristiken zum Optimieren von SPARQL-Anfragen definiert, um für eine Anfrage einen möglichst optimalen Ausführungsplan zu erstellen. Im Kontext der Anfragebearbeitung wurden auch graphmusterbasierte Indexe spezifiziert und deren Nutzen für die Verarbeitung von Anfragen untersucht.

Das RCS-Speichermodell wurde prototypisch implementiert und im Vergleich zum nativen RDF-DBMS Jena TDB evaluiert. Die durchgeführten Experimenten zeigen, dass auf Grund des Vermeidens von kostenintensiven Join-Operationen das RCS-Speichermodell insbesondere für das Beantworten von Anfragen mit großen sternförmigen Teilmustern geeignet ist.

Inhaltsverzeichnis

Kapitel 1

Einführung

Das Thema dieser Arbeit ist die Entwicklung eines nativen Speichermodells für das Verwalten und Anfragen von RDF-Daten und dessen Evaluation mittels einer prototypischen Implementierung, dem Resource Centered Store (RCS).

Abschnitt 1.1 beschreibt die Motivation für die Entwicklung eines neuartigen Speichermodells für RDF-Daten. Im Abschnitt 1.2 werden die Beiträge dieser Arbeit aufgelistet. Schließlich wird im Abschnitt 1.3 ein Überblick über die Inhalte der Kapitel gegeben.

1.1 Motivation

Mit den beiden W3C-Spezifikationen [KC04] und [HPS14] wurden die konzeptionellen Grundlagen für das Resource Description Framework (RDF) definiert, um die Eigenschaften von und die Beziehungen zwischen realen und virtuellen Objekten – so genannten Ressourcen – maschinenverarbeitbar zu beschreiben. Mit Hilfe von RDF können Maschinen unter anderem automatisch Daten zu einer Ressource lokalisieren und verarbeiten, unterschiedliche Bedeutungen einer Zeichenkette erkennen und implizite Informationen ableiten.

Die wesentlichen Merkmale des Resource Description Frameworks sind:

- Das Datenmodell basiert auf gerichteten und beschrifteten Multigraphen (*engl. directed labeled multigraph*).
- Ressourcen und Eigenschaften werden eindeutig durch IRIs (Internationalized resource identifier) identifiziert. Im Folgenden wird ein solcher Multigraph als *RDF-Graph* bezeichnet.
- Eigenschaften sind selbst wieder Ressourcen und es können über diese Informationen in einem RDF-Graph bereitgestellt werden.

Aufgrund dieser Eigenschaften konnten die im relationalen und objektorientierten Datenbankkontext definierten Anfragesprachen nicht ohne Weiteres verwendet werden. Noch bevor RDF den Status eine W3C Recommentation erreichte, wurden unabhängig vom W3C spezialisierte Anfragesprachen wie beispielsweise RQL [KAC+02] und SquishQL [MSR02] definiert, um der

Graphstruktur der RDF-Daten Rechnung zu tragen. Einige Jahre später griff das W3C einige Vorschläge auf und begann eine Anfragesprache für RDF zu entwickeln. Im Jahr 2010 wurde die Version 1.0 und 2013 die Version 1.1 der *SPARQL Query Language for RDF* [PS08] als W3C Recommendations veröffentlicht.

Das wesentliche Charakteristikum von SPARQL ist ein deklaratives Beschreiben von Teilgraphen, wobei anstelle von konkreten Ressourcen und Eigenschaften auch Variablen verwendet werden können. Diese Art von Anfragen werden im Kontext von SPARQL als *Graphmuster* bezeichnet. Die Herausforderungen für RDF-Datenbanksysteme bestehen im Lokalisieren aller einem Graphmuster entsprechenden Teilgraphen in ein oder mehreren RDF-Graphen.

Im Laufe der letzten Jahre der Forschung im Bereich des RDF-Datenmanagements wurde festgestellt, dass existierende Datenbankmanagementsysteme nur unzureichend die Charakteristika des RDF-Datenmodells und von SPARQL unterstützen (vgl. Abschnitt 3.1). Relationale Datenbankmanagementsysteme (DBMS) sind zum Speichern und Anfragen von graphartigen Strukturen ungeeignet, da entweder viele kostenintensive Join-Operationen zum Berechnen der Lösungen eines Graphmusters ausgeführt werden müssen oder die dem RDF-Datenmodell inhärente Flexibilität des Schemas verloren geht. Obwohl das Datenmodell von objektorientierte DBMS sehr ähnlich zum dem RDF-Datenmodell ist, sind diese zum Verwalten von RDF-Daten ungeeignet: Im OO-Datenmodell sind Eigenschaften keine Entitäten erster Klasse, d.h. sie können selbst nicht als Objekte verwendet werden. Darüber hinaus erfordern deren Zugriffsstrukturen ein fest definiertes Schema. Graphdatenbanksysteme sind für die Verwaltung von RDF-Graphen aufgrund ihres Fokus ungeeignet. Deren Hauptanwendungsgebiet ist das Lokalisieren die zu einem Anfragegraphen isomorphen (Teil-)Graphen in einer großen Menge von Graphen, die nicht miteinander verknüpft sind.

Aus diesen Gründen werden seit einigen Jahren speziell für das RDF-Datenmodell und SPARQL entworfene Datenmanagementsystem entwickelt und erforscht. Die performantesten RDF-DBMS basieren auf dem Indexieren der einzelnen Aussagen mittels mehrerer B+-Bäume[1] (z. B. [WKB08, NW10]). Da jede Aussagen einzeln indexiert wird, müssen ähnlich wie bei relationalen DBMS Join-Operationen ausgeführt werden, um die Lösungen eines Graphmuster zu berechnen.

Mit dem in dieser Arbeit entwickelten RCS-Speichermodell werden die Vorteile der relationalen DBMS-Architektur und von nativen RDF-DBMS zusammengeführt:

I) Jede Aussage wird nur ein einziges Mal in der Datenbank abgespeichert.

II) Aus relationalen DBMS bekannte Algorithmen und Verfahren können in den RCS integriert werden.

III) Die Anzahl der Join-Operationen zum Berechnen von Lösungen eines Graphmusters wird reduziert.

[1] Bei Unterstützung von benannten Graphen werden mindestens sechs Indexe erzeugt (vgl. Abschnitt 3.1.2).

IV) Das System kann mit wenig Aufwand um zusätzliche Zugriffsstrukturen erweitert werden. Dies ermöglicht das Indexieren von zur Zeit der Entwicklung des RCS unbekannten Inhalten (z. B. Indexieren des Volltexts der Literalen).

1.2 Ergebnisse & Lösungen der Arbeit

In dieser Arbeit wird der Resource Centered Store (RCS) präsentiert, der ein neuartiges Speichermodell zur Verwaltung und Indexierung von RDF-Daten realisiert. Das Speichermodell kombiniert dabei Charakteristiken des graphartigen RDF-Datenmodells mit in relationalen Datenbanksystemen etablierten Konzepten zur Datenverwaltung.

Die folgenden Beiträge werden in den Kapiteln dieser Arbeit erbracht:

- **RCS-Speichermodell.**
  Es wird ein neuartiges Speichermodell zur Verwaltung von RDF-Daten auf Datenseiten präsentiert, das auf der Gruppierung der Tripel anhand ihres Subjekts basiert. Bei gegebenem Subjekt erlaubt das Speichermodell die Auswertung von sternförmigen Anfragen mit konstanter und bei einer Variablen als Subjekt mit linear Komplexität. Insbesondere ist die Komplexität unabhängig von der Anzahl der Tripelmuster. In Kapitel 3 wird das Speichermodell inklusive Zugriffsstrukturen und Operationen beschrieben.

- **Graphmuster-basierte Indexierung.**
  Um neben sternförmigen Anfragen auch Pfadanfragen effizient auswerten zu können, wird in Kapitel 4 ein Konzept zur graphmuster-basierten Indexierung von RDF-Daten entwickelt. Die Grundlage bildet dabei das Materialisieren von Basis-Graphmustern und das Einbeziehen der Indexe in die Anfrageauswertung.

- **SPARQL Query Graph Model.**
  Zur Repräsentation und Optimierung einer Anfrage in der Anfrageverarbeitungskomponente wird in Kapitel 5 das SPARQL Query Graph Modell (SQGM) entwickelt. Die Grundlage für dieses Modell formt das in [HCL+90] beschriebene Query Graph Model für relationale Datenbanksysteme.

- **Algorithmen und Transformationsregeln.**
  Um die besonderen Charakteristika des RCS-Anfragemodells wie den effizienten Zugriff auf alle Tripel mit demselben Subjekt ausnutzen zu können, werden in Kapitel 5 spezialisierte Operatoren und Algorithmen entwickelt. Darüber hinaus werden entsprechende Transformationsregeln für SPARQL Query Graph Model definiert.

- **Experimentelle Evaluation.**
  Das RCS-Speichermodell und das Index-Konzept wurden basierend auf dem Jena2 RDF-Framework prototypisch implementiert. In den in Kapitel 6 beschriebenen Experimenten konnte insbesondere gezeigt werden,

dass die Beantwortung von Anfragen mit gegebenem Subjekt mit konstanter Komplexität erfolgt und dass bei beliebigen sternförmigen Anfragen größere Graphmuster sich vorteilhaft auf die Ausführungszeit auswirken.

In einigen RDF-Vokabularen (z. B. RDF Schema [BG04] und Web Ontology Language [W3C12]) sind unter anderem auch Regeln definiert, mit deren Hilfe neue Aussage aus einem RDF-Graphen hergeleitet werden können. Ebenso kann mittels Regeldefinitionssprachen (z. B. Semantic Web Rule Language [HPSB+04]) eine Menge von Regeln spezifiziert werden, um implizit Aussagen zu erzeugen. In dieser Arbeit werden Anfragen auf derartig erweiterte RDF-Graphen nicht betrachtet.

1.3 Struktur der Arbeit

Abbildung 1.1 illustriert den allgemeinen Aufbau eines RDF-DBMS, wobei die in dieser Arbeit betrachteten und implementierten Komponenten rot hervorgehoben wurden. Diese Arbeit ist wie folgt strukturiert:

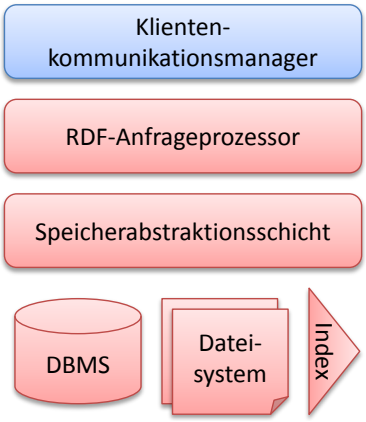


Abbildung 1.1: Generische Architektur eines RDF-DBMS

In Kapitel 2 werden zunächst grundlegenden Konzepte und Notationen bezüglich des RDF-Datenmodells eingeführt und definiert. Kapitel 3 befasst sich mit der Speicherabstraktionsschicht und der Datenverwaltung (DBMS und Dateisystem). Es führt das RCS-Speichermodell zum Speichern von RDF-Graphen ein und definiert darauf grundlegende Operationen. Das Kapitel schließt mit einer theoretischen Komplexitätsanalyse dieser Operationen. Anschließend wird im Kapitel 4 ein Konzept zur graphmuster-basierten Indexierung von RDF-Daten präsentiert (vgl. Komponente Index in Abbildung 1.1). Dieser Ansatz ist mit materialisierten Sichten in relationalen Datenbanken vergleichbar. Wenn ein Graphmuster indexiert wird, werden die Lösungen vorberechnet und gespeichert. Diese können während der Anfragebearbeitung zur Berechnung von Teillösungen einer Anfrage herangezogen werden.

Aufbauend auf den Konzepten zur Verwaltung und Indexierung von RDF-Daten befasst sich das Kapitel 5 mit der Verarbeitung von SPARQL-Anfragen und beschreibt für diese Algorithmen für Operatoren sowie Optimierungsstrategien (Komponente RDF-Anfrageprozessor). Kapitel 6 beschreibt die prototypische Implementierung des RCS-Speichermodells und der graphmusterbasierte Indexierung und präsentiert die Erkenntnisse der experimentellen Evaluation. Kapitel 7 fasst die in der Arbeit gewonnenen Erkenntnisse zusammen und gibt einen Ausblick auf mögliche zukünftige Arbeiten.

Kapitel 2

Grundlagen

Die beiden W3C-Spezifikationen [KC04] und [HPS14] beschreiben die konzeptionellen Grundlagen, die Syntax und die Semantik des Resource Description Frameworks (RDF). Bei der Definition des Datenmodells für RDF wurde unter anderem auf dessen Einfachheit wert gelegt. In RDF werden Informationen als gerichtete, beschriftete Multigraphen (*engl. directed labeled multigraph*) repräsentiert. Im Folgenden werden diese als RDF-Graph oder kurz als Graph bezeichnet. Zwei Knoten und eine Kante zwischen diesen stellen in RDF eine Informationseinheit dar, sie formen eine sogenannte Aussage (*engl. statement*). Somit besteht ein RDF-Graph aus einer Menge von Aussagen. Im Gegensatz zu anderen Modellen (z. B. ER-Modell [Che76]) zur Repräsentation von Beziehungen zwischen Objekten können in RDF ausschließlich binäre Beziehungen dargestellt werden. Darüber hinaus wird in RDF nicht angenommen, dass alle Aussagen zu einer Ressource zum Zeitpunkt der Anfrage bekannt sind. Durch die Eindeutigkeit der IRIs könnten weitere Aussagen zu einer Ressource in anderen RDF-Datenquellen existieren (Open-World-Assumption) [KC04]

Neben der Datenstruktur definiert sich ein Datenmodell auch durch seine Operationen und Inferenzregeln über den Daten sowie einer Menge generischer Integritätsbedingungen [Cod80]. Das Resource Description Framework bildet nur die Grundlage für eine Reihe von darauf aufbauenden Spezifikationen. Beispielsweise werden in den Spezifikationen RDF-Schema [BG04] und OWL [W3C12] Vokabulare sowie Inferenzregeln definiert. Mit Hilfe der Inferenzregeln lassen sich zusätzliche Aussagen aus einem gegebenen RDF-Graph ableiten.

Zur Extraktion von Informationen aus einem RDF-Graph wurden zunächst eine Reihe von Anfragesprachen entwickelt (z. B. RQL [KAC+02], SquishQL [MSR02]), die die Grundlage für die Spezifikation der *SPARQL Query Language for RDF* [PS08] bildeten. Diese Spezifikation erlaubte zunächst nur die Extraktion jedoch keine Modifikation von Aussagen eines RDF-Graphen. Erst im Rahmen der Weiterentwicklung der Anfragesprache SPARQL zur Version 1.1 [PS13] wurden Operationen zur Modifikation eines RDF-Graphen in der Spezifikation *SPARQL 1.1 Update* [GPP13] definiert. Die spezifizierten Operationen erlauben das Hinzufügen und Löschen von RDF-Graphen und Aussagen.

Da diese Arbeit die Speicherung und das Anfragen von RDF-Daten fokussiert, wird im Folgenden nur auf das RDF-Datenmodell und die Anfragespra-

che SPARQL detailliert eingegangen. In diesem Abschnitt wird jedoch nicht näher auf SPARQL Update eingegangen, da die darin definierten Operationen auf das Hinzufügen und Löschen von Aussagen sowie das Anfragen eines RDF-Graphen reduziert werden kann. Diese Operationen werden im Anschluss an der Beschreibung des in dieser Arbeit entwickelten Speichermodells diskutiert.

Im ersten Abschnitt 2.1 werden die wesentlichen Eigenschaften des RDF-Datenmodells vorgestellt und die in der Arbeit verwendeten Notationen für RDF eingeführt. Im anschließenden Abschnitt 2.2 werden die für diese Arbeit relevanten Begriffe der Anfragesprache SPARQL beschrieben.

2.1 RDF-Datenmodell

In der Spezifikation [HPS14] wird das Datenmodell zur Repräsentation von Informationen als Multigraph definiert. Die zentralen Elemente im Resource Description Framework sind *Ressourcen* (*engl. resource*) und deren *Eigenschaften* (*engl. property*) sowie *Literale* (*engl. literal*).

Unter einer Ressource versteht man einen Stellvertreter für elektronische Artefakte (z. B. Dokumente und Bilder) aber auch für Objekte der realen Welt (z. B. Gegenstände und Personen). Ein wesentlicher Aspekt des RDF-Datenmodells ist, dass jede Ressource eindeutig über eine *IRI-Referenz* (*engl. IRI reference*) identifiziert wird.

Eigenschaften sind eine Teilmenge der Menge der Ressource und werden dazu verwendet die Charakteristika von Ressourcen näher zu beschreiben. Im Vergleich zu anderen Datenmodellen sind Eigenschaften im RDF-Datenmodell Objekte erster Klasse (*engl. first-class citizen*). Damit ist es auch möglich, Eigenschaften mittels Eigenschaften näher zu beschreiben.

Darüber hinaus werden im Standard auch sogenannte *anonyme Ressourcen* (*engl. blank nodes*) definiert, die nicht durch eine IRI in einem RDF-Graph identifiziert werden. Diese wurden eingeführt, um einen Platzhalter für Ressourcen zu haben, die selbst nicht von Interesse sind. Im Gegensatz zu Ressourcen können anonyme Ressourcen nicht direkt adressiert werden, sondern können nur über die Spezifikation von ihren Eigenschaften aus einem RDF-Graph extrahiert werden. Innerhalb eines RDF-Graphen sind anonyme Ressourcen jedoch wohl unterschieden. Der Nutzen von anonymen Ressourcen für die Qualität von RDF-Daten wird inzwischen bezweifelt. Beispielsweise schreiben Heath und Bizer in [HB11], dass sie im Kontext von Linked Data ein Referenzieren und Verknüpfen von Daten verhindern.

Für textuelle und nummerische Werte in RDF wird der Begriff Literal (*engl. literal*) verwendet. In seiner einfachsten Form ist ein Literal eine Zeichenkette und hat eine fest, der Zeichenkette inhärente Bedeutung. Neben dieser Form kann ein Literal auch mit einem Typ (z. B. dem XML-Schema-Datentyp `xsd:integer`) versehen und damit Einfluss auf dessen Interpretation genommen werden. Ein mit einem Datentyp versehenes Literal wird als *getyptes Literal* (*engl. typed literal*) bezeichnet. Für die Arbeit ist die Unterscheidung zwischen getypten und ungetypten Literal unerheblich, weshalb im Folgenden nur von Literalen gesprochen wird.

Die nachfolgende Definition 2.1 gibt eine Übersicht über die eingeführten Mengen, wobei IRI-Referenz, Literal und anonyme Ressourcen wie in [KC04] definiert sind:

Definition 2.1. *Die Menge der* IRI-Referenzen *wird mit* $\mathbb{I}$, *die Menge der* Literale *mit* $\mathbb{L}$ *und die Menge der* anonymen Ressourcen *mit* $\mathbb{B}$ *bezeichnet. Insbesondere sind die Mengen paarweise disjunkt:*

$$\mathbb{I} \cap \mathbb{L} = \emptyset, \quad \mathbb{I} \cap \mathbb{B} = \emptyset, \quad \mathbb{B} \cap \mathbb{L} = \emptyset$$

Die Menge bestehend aus allen Ressourcen und Literalen, $\mathbb{I} \cup \mathbb{B} \cup \mathbb{L}$, wird als die Menge der *RDF-Terme* bezeichnet und mittels des Symbols RDF-T referenziert.

Beispiel 2.1. Die IRI `<http://xmlns.com/foaf/0.1/Person>` identifiziert eine Ressource. Wenn der Namespace einer IRI geläufig ist, wird im Folgenden die abgekürzte Namespace-Schreibweise verwendet (z. B. `foaf:Person`). Die Zeichenketten `"Hans Schmidt"` sowie `"77"^^xsd:integer` sind Literale. □

Mit Hilfe von Ressourcen, Eigenschaften und Literalen können Aussagen (*engl. statements oder triple*) gebildet werden, mit denen durch eine IRI referenzierte Objekte näher beschrieben werden können. Eine Aussage hat die Form eines Tripels `s p o`, wobei dessen Komponenten respektive als Subjekt, Prädikat und Objekt (*engl. subject*, *predicate und object*) bezeichnet werden. Eine (anonyme) Ressource wird durch ein Tripel näher beschrieben, indem diesem eine Eigenschaft mit einer (anonymen) Ressource oder einem Literal als Wert zugeordnet wird. In Definition 2.2 werden Tripel formal definiert.

Definition 2.2 (Tripel)**.** *Eine Tripel (Aussage) ist ein Element aus* $(\mathbb{I} \cup \mathbb{B}) \times \mathbb{I} \times (\mathbb{I} \cup \mathbb{B} \cup \mathbb{L})$. *Die Menge aller Tripel wird mit* $\mathcal{T}$ *bezeichnet.*

Für ein Tripel t werden die Notationen t^s, t^p und t^o als Referenz für das Subjekt, Prädikat und Objekt vereinbart.

Zum Notieren von Aussagen wurden unterschiedliche Formate entwickelt. Die bekanntesten sind die RDF/XML-Syntax [Bec04], N3-Notation [BL06] und Turtle [Bec07]. Aufgrund ihrer kompakten Notation wird in dieser Arbeit die Turtle-Notation verwendet. IRIs werden wie zuvor eingeführt entweder in spitzen Klammern gesetzt oder es wird die Namensraumschreibweise verwendet. Für anonyme Ressourcen wird der Namensraum _ verwendet (z. B. `_:bn`). Da die Turtle-Notation gut verständlich ist, wird nicht auf weitere syntaktische Formen eingegangen, sondern bei Unklarheiten auf [Bec07] verwiesen.

Beispiel 2.2. Wenn beispielsweise die IRIs `ex:pers1` und `ex:pers2` die zwei Personen Person 1 und Person 2 identifizieren, dann drücken die beiden folgenden beiden Tripel aus, dass eine Person 1 die Person 2 kennt und Person 1 den Namen „Hans Schmidt" hat:

```
ex:pers1 foaf:knows ex:pers2.
ex:pers1 foaf:name "Hans Schmidt".
```
□

Neben der textuellen Repräsentation von Aussagen wird in der RDF-Spezifikation auch eine graphische Darstellung definiert. Ressourcen werden als Ellipsen und Literale als Rechteck dargestellt, wobei diese mit der IRI bzw.

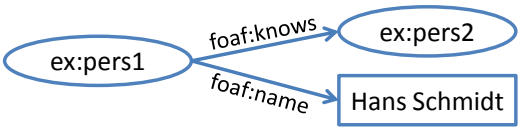


Abbildung 2.1: Beispiel eines RDF-Graphen

dem Literal beschriftet sind. Eigenschaften verbinden als gerichtete, beschriftete Kanten Ressourcen und Literale untereinander. Abbildung 2.1 zeigt die Repräsentation des RDF-Graphen aus Beispiel 2.2.

Wenn man nicht nur ein einzelnes Tripel sondern eine Menge von Tripeln betrachtet, so erhält man einen RDF-Graph.

Definition 2.3 (RDF-Graph)**.** *Eine Menge von Tripeln $G \subseteq \mathcal{T}$ wird als RDF-Graph bezeichnet.*

Aus den bisher gegebenen Definitionen ergibt sich das RDF-Datenmodell als die Struktur eines beschrifteten, gerichteten Multigraphen. Dabei werden Ressourcen und Literale als Knoten und Eigenschaften als gerichtete Kanten interpretiert, wobei mehrere Kanten (mit verschiedenen Beschriftungen) zwischen zwei konkreten Knoten erlaubt sind. Zwei durch eine Kante verbundenen Knoten entsprechen dabei einer Aussage. Sowohl Knoten als auch Kanten sind mit den jeweiligen IRIs bzw. Literalen beschriftet, die Knoten von anonymen Ressourcen haben keine Beschriftung. Als Ergebnis der eindeutigen Identifizierbarkeit von Ressourcen anhand ihrer IRI gibt es in dem entstehenden Modell keine zwei Knoten mit derselben Beschriftung/IRI. Da es sich bei einem RDF-Graph um eine Menge von Tripeln handelt, ist darüber hinaus jedes Tripel nur einmalig in einem RDF-Graph enthalten.

Benannte Graphen Mit den bisherigen Definitionen lassen sich Ressourcen eindeutig identifizieren und Aussagen über diese formulieren. Über die Verwendung eines eigenen Namensraumes können Ressourcen und Eigenschaften zu einem Vokabular zusammengefasst werden. Darüber hinaus ist in der RDF-Spezifikation auch vorgesehen, dass man Aussagen über Aussagen treffen kann (Reifikation). Obwohl Aussagen zu einem Dokument zusammengefasst und über eine IRI im Internet zugegriffen werden können, ist es auf Basis von des Resource Description Frameworks nicht möglich, über einen RDF-Graph bzw. Namensraum Aussagen zu tätigen.

Um diesem Problem zu begegnen, wurde von Carrol et al. das Konzept der benannten Graphen (*engl. Named Graph*) entwickelt [CBHS05], mit dem eine Menge von Aussagen mit einer IRI benannt werden kann. Diese IRI lässt sich dann in Aussagen verwenden, um beispielsweise Metadaten zu einem RDF-Graph definieren zu können. Die Benennung von Graphen mit IRIs findet unter anderem auch in der Anfragesprache SPARQL Verwendung. In der Literatur wird oftmals auch der Begriff Kontext (*engl. context*) als Synonym für einen benannten Graph benutzt.

Definition 2.4. *Sei $\mathcal{P}_\mathcal{T}$ die Potenzmenge über die Menge aller Tripel $\mathcal{T}$. Ein benannter Graph ist ein geordnetes Paar $ng = (n, g)$ mit $n \in \mathbb{I}$ und $g \in \mathcal{P}_\mathcal{T}$.*

Die IRI n eines benannten Graphen $ng = (n, g)$ wird im Folgenden auch als Name des Graphen bezeichnet.

Ähnlich zu der eindeutigen Identifizierbarkeit von Ressourcen durch IRIs gilt auch für benannte Graphen, dass es keine zwei Graphen mit demselben Namen geben kann. Jedoch können zwei Graphen mit unterschiedlichem Namen Aussagen über dieselbe Ressource beinhalten.

Notationen In der RDF-Spezifikation wird neben dem Datenmodell auch ein einfaches Vokabular definiert, um grundlegende Eigenschaften einer Ressource spezifizieren zu können. Eine wesentliche Eigenschaft einer Ressource ist ihr Typ (`rdf:type`), deren Wert wiederum eine Ressource ist. Einer Ressource können mehrere Typen zugeordnet werden. Um auf einfache Art den Typ einer Ressource referenzieren zu können, wird dafür das Symbol τ eingeführt. Folgende Notationen werden in dieser Arbeit verwendet:

- $\tau(r) = \tau_t$ bezeichnet die Typen einer Ressource.
- $|\tau_t|$ referenziert die Anzahl der Ressourcen vom Typ t.
- $\tau(G) = \{r \in G : \exists t \in G : \tau(t^s) = r\}$ ist die Menge aller Ressourcen, die als Typ des Subjekts irgendeines Tripels aus einem RDF-Graph G verwendet wird.
- $|\tau(G)|$ referenziert die Anzahl der unterschiedlichen Ressource-Typen in einem Graph G.

2.2 Anfragesprache SPARQL

In [PS13] wird die Anfragesprache SPARQL für RDF in der überarbeiteten und erweiterten Version 1.1 beschrieben. Neben zusätzlichen Sprachkonstrukten und Standardfunktionen wurden insbesondere Operationen zum Modifizieren von RDF-Graphen eingeführt.

Um aus einem RDF-Graph Informationen zu extrahieren, werden die diese Information Teilgraphen beschrieben. Die wichtigsten Sprachkonstrukte zum Beschreiben der Teilgraphen sind Tripelmuster, Basis-Graphmuster und Filterausdrücke. Auf diesen aufbauend werden in der SPARQL-Spezifikation weitere Konstrukte definiert. In diesem Abschnitt wird jedoch kein umfassender Überblick über die Konzepte von SPARQL geben, sondern nur die für diese Arbeit relevanten eingeführt; eine Beschreibung aller Sprachkonstrukte kann auf [PS13] nachgelesen werden.

Zunächst werden die grundlegenden Bausteine einer Anfrage beschrieben. Anschließend werden in Abschnitt 2.2.2 die Lösungen eines Basis-Graphmusters bezüglich eines RDF-Graphen definiert. In Abschnitt 2.2.3 wird kurz auf das Kombinieren von Basis-Graphmustern zu komplexeren Strukturen eingegangen. Anschließend wird das Konzept von sternförmigen Graphmustern beschrieben (Abschnitt 2.2.4).

2.2.1 Sprachkonstrukte

Tripelmuster (*engl. triple pattern*) sind zu Tripeln ähnlich definiert und formen die kleinste Einheit, um einen Teilgraphen zu beschreiben. Im Vergleich zu

Tripeln sind in einem Tripelmuster neben Ressourcen und Literalen auch Variablen in allen Komponenten erlaubt.

Definition 2.5 (Tripelmuster)**.** *Sei mit* $\mathbb{V}$ *die Menge aller Variablen bezeichnet. Ein* Tripelmuster *ist ein Element aus* $(\mathsf{RDF\text{-}T} \cup \mathbb{V}) \times (\mathbb{I} \cup \mathbb{V}) \times (\mathsf{RDF\text{-}T} \cup \mathbb{V})$*. Die Menge aller Tripel wird als* $\mathcal{T}$ *notiert.*

Syntaktisch werden Variablen in dieser Arbeit durch ihren Namen mit vorangestelltem '?' dargestellt (z. B. `?var`).[1] Ohne der Definition der Lösung eine Anfrage vorweggreifen zu wollen, sein an dieser Stelle angemerkt, dass Variablen an konkrete Werte (Ressourcen oder Literale) eines RDF-Graphen gebunden werden. Die Zuordnung eines RDF-Terms zu einer Variablen wird mit Variablenbindung (*engl. variable binding*) bezeichnet.

Zum Beschreiben größerer Teilgraphen werden mehrere Tripelmuster zu einem Basis-Graphmuster (*engl. basic graph pattern*) zusammengefasst.

Definition 2.6 (Basis-Graphmuster)**.** *Ein* Basis-Graphmuster *besteht aus einer Menge von Tripelmustern. Die Menge aller Basis-Graphmuster wird mit* BGP *bezeichnet.*

An dieser Stelle werden ein paar Notationen eingeführt. Sei $P \in \mathsf{BGP}$ ein Basis-Graphmuster, dann bezeichnet $|P|$ die Anzahl der Tripelmuster in P. Darüber hinaus bezeichne $t^{s=r} = \{t \in G : t^s = r\}$ die Menge der Tripelmuster mit dem Subjekt r; für Prädikat und Objekt sind die Mengen $t^{s=p}$ und $t^{s=o}$ analog definiert.

Eine Verknüpfung der verschiedenen Tripelmuster erfolgt über deren Ressourcen, Literale und Variablen. Das heißt, wenn zwei Tripelmuster dieselbe Variable beinhalten, dann werden sie innerhalb einer Lösung an denselben Wert gebunden.[2] Die zwei Tripelmuster oder Basis-Graphmuster verknüpfende Variablen werden als *Join-Variablen* bezeichnet.

Abbildung 2.2 zeigt beispielsweise ein Basis-Graphmuster bestehend aus zwei Tripelmustern, die über die Variable `?p` miteinander verknüpft sind. Es wird nach allen Ressourcen gefragt, die mit der Ressource `ex:pers2` über die Eigenschaft `foaf:knows` und mit einem RDF-Term über `foaf:name` in Beziehung stehen.

```
?p foaf:knows ex:pers2 .
?p foaf:name ?name .
FILTER (?name = "Karl")
```

Abbildung 2.2: Beispiel eines Basis-Graphmuster mit Filterausdruck

Durch das Kombinieren von Basis-Graphmustern lassen sich komplexer strukturierte Teilgraphen in einer Anfrage spezifizieren. Auf diesen Punkt wird in Abschnitt 2.2.3 näher eingegangen, da die Kenntnis über die Lösung eines Basis-Graphmusters das Verständnis vereinfacht.

In einer SPARQL-Anfrage lassen sich über *Filterausdrücke* zusätzlich die Teilgraphen einschränken, die für die Berechnung ihrer Lösungen berücksichtigt werden. Filterausdrücke sind durch das Schlüsselwort `FILTER` gekennzeichnet (vgl. Abbildung 2.2) und können neben den wohl bekannten Ver-

[1] Laut Spezifikation [PS13] kann anstelle von '?' auch das Zeichen '$' verwendet werden.

[2] Das Verknüpfen von Tripelmuster verhält sich ähnlich zu Join-Operationen in relationalen Datenbankmanagementsystemen (RDBMS).

gleichsoperatoren auch vordefinierte Funktionen (z. B. `isLiteral()`) beinhalten. Wenn für ein Basis-Graphmuster eine Menge von Variablenbindungen bestimmt worden ist, so werden durch einen Filterausdruck diejenigen Variablenbindungen eliminiert, für die dieser Ausdruck zu falsch evaluiert.

2.2.2 Lösung eines Basis-Graphmusters

Basierend auf Spezifikation [PS13] werden die für diese Arbeit wesentlichen Definitionen eingeführt. Insbesondere wird der Fokus auf Lösungen für ein Basis-Graphmuster gelegt, da deren Lösungen die Grundlage für das Berechnen der Gesamtlösung einer Anfrage bilden. Da an dieser Stelle die SPARQL-Spezifikation nicht repliziert werden soll, wird der Leser für tiefer gehende Details auf [PS13] verwiesen.

Intuitiv kann die Berechnung der Lösungen zu einer SPARQL-Anfrage wie folgt beschrieben werden: Für ein Basis-Graphmuster der Anfrage wird für jede Variable nach einem RDF-Term gesucht, so dass nach dem Ersetzen der Variablen mit den ihnen zugeordneten RDF-Termen ein Teilgraph entsteht, der im RDF-Graph enthalten ist. Die Multimenge aller dieser Abbildungen formen die Lösungen eines Basis-Graphmusters.

Zunächst wird die Abbildung von Variablen auf RDF-Terme eingeführt, die sogenannte Lösungsabbildung (*engl. solution mapping*).

Definition 2.7 (Lösungsabbildung)**.** *Eine* Lösungsabbildung *ist eine partielle Funktion der Form* $\mu : \mathbb{V} \rightarrow \mathsf{RDF\text{-}T}$.

Für ein Basis-Graphmuster $P \in \mathsf{BGP}$ und eine Lösungsabbildung μ bezeichnet $\mu(P)$ das Graphmuster, dass durch das Ersetzen einer jeden Variable $v \in \mathrm{dom}(\mu)$ wird ihren Wert $\mu(v)$ entsteht. Wenn alle Variablen in P durch RDF-Terme ersetzt werden, dann wird μ im Folgenden als *vollständig* bezeichnet.

Wie eingangs informell beschrieben, wird also zum Berechnen der Lösungen für ein Basis-Graphmuster P nach vollständigen Lösungsabbildungen μ gesucht, so dass $\mu(P)$ im RDF-Graph enthalten ist. Anonyme Ressourcen spielen beim Validieren, ob $\mu(P)$ in einem RDF-Graphen enthalten ist, eine besondere Rolle. Der Grund dafür ist, dass diese für beliebige RDF-Terme stehen können – also auch Ressourcen und Literale. Mit der folgenden Definition werden RDF-Instanzabbildungen (*engl. RDF instance mapping*) eingeführt.

Definition 2.8 (RDF-Instanzabbildung)**.** *Eine* RDF-Instanzabbildung *ist eine Abbildung der Form* $\sigma : \mathbb{B} \rightarrow \mathsf{RDF\text{-}T}$.

Im Grunde genommen nehmen anonyme Ressourcen die Rolle von Variablen ein, jedoch kann im Gegensatz zu Variablen im Ergebnis einer Anfrage nicht auf den zugeordneten Wert zugegriffen werden. Durch Anwenden einer RDF-Instanzabbildung auf einem RDF-Graph G erhält man einen neuen RDF-Graph G'. Der so entstandene Graph G' wird als Instanz von G bezeichnet. Die Instanz-sein-Beziehung ist reflexiv und transitiv. [HPS14]

Als Notation wird vereinbart, dass $\mu[P]$ bzw. $\sigma[P]$ das Anwenden der jeweiligen Abbildung auf das Graphmuster P bezeichnen.

Die eingangs gegebene intuitive Definition einer Lösung kann nun verfeinert werden. Die Lösungen für ein Basis-Graphmuster P bezüglich eines RDF-Graphen G werden berechnet, indem ein Graph durch das Einsetzen von Variablen und anonyme Ressourcen im Graphmuster durch RDF-Terme gebildet

wird. Wenn der so entstandene Graph ein Teilgraph von G ist, dann ist die durch μ gegebenen Variablenbindungen eine Lösung von P. Die folgende Definition formalisiert das Finden von Lösungen eines Basis-Graphmusters.

Definition 2.9 (Lösung eines Basis-Graphmusters)**.** *Sei G ein RDF-Graph und $P \in \mathsf{BGP}$ ein Basis-Graphmuster. μ ist eine Lösung für das Graphmuster P über G, wenn es eine RDF-Instanzabbildung σ gibt, so dass $\mu \circ \sigma[P] = \sigma[\mu[P]] \subseteq G$ ist und μ auf die Variablen von P eingeschränkt ist.*

Wenn es für ein Basis-Graphmuster P eine Lösung bezüglich eines RDF-Graphen G gibt, dann wird im Folgenden die Sprechweise *P kommt in G vor* verwendet. Die Menge der Lösungen für das Basis-Graphmuster P wird mit Ω_P^G bezeichnet. Wenn von dem Kontext her der RDF-Graph klar ist, wird nur Ω_P geschrieben. Für die Kardinalität einer Lösungsmenge wird die Schreibweise $|\Omega_P|$ verwendet. Falls die Ordnung der Lösungen eines Basis-Graphmusters von Relevanz ist (z. B. bei der Definition von Operatoren), dann bezeichnet Ψ eine Sequenz von Lösungen.

Beispiel 2.3. Abbildung 2.3 zeigt auf der linken Seite ein Basis-Graphmuster P und auf der rechten Seite einen RDF-Graph G. Für die Lösungsabbildung μ : $\{?p \rightarrow \texttt{ex:pers1}; ?name \rightarrow \text{"}\texttt{Karl}\text{"}\}$ gibt eine RDF-Instanzabbildung $\sigma : \_:\texttt{bn} \rightarrow ex : pers2$, so dass $\sigma[\mu[P]]$ ein Teilgraph von G ist. Daher ist μ eine Lösung von P über G. □

```
?p foaf:knows _:bn .      ex:pers1 foaf:knows ex:pers2 .
?p foaf:name ?name .      ex:pers1 foaf:name "Karl" .
                          ex:pers1 foaf:mbox "karl@ex.com" .
```

Abbildung 2.3: Basis-Graphmuster P (links) und RDF-Graph G (rechts)

Falls die Lösungen für ein Graphmuster bestehend aus mehreren Basis-Graphmustern berechnet werden sollen, so müssen die Lösungen der Basis-Graphmuster miteinander kombiniert werden. Zwei Lösungsabbildungen können miteinander kombiniert werden, wenn sie sich nicht widersprechen und *kompatibel* miteinander sind. Das heißt, die beiden zugehörigen Lösungsabbildungen bilden dieselben Variablen auf dieselben RDF-Terme ab. (vgl. [PS13] und [AGP10])

Definition 2.10 (Kompatible Lösungsabbildungen)**.** *Zwei Lösungsabbildungen μ_1 und μ_2 sind miteinander* kompatibel*, wenn das Folgende gilt:*

$$\forall x \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2) : \mu_1(x) = \mu_2(x)$$

Aus Definition 2.10 lässt sich ableiten, das die Vereinigung zweier Lösungsabbildungen $\mu_1 \cup \mu_2$ wiederum eine Lösungsabbildung ist. In [PS13] wird hierfür auch die Notation $merge(\mu_1, \mu_2)$ eingeführt.

Beispiel 2.4. Gegeben seien die folgenden drei Lösungsabbildungen:

- $\mu_1 : ?p \rightarrow \texttt{ex:pers1}; ?name \rightarrow \text{"}\texttt{Karl}\text{"}$
- $\mu_2 : ?p \rightarrow \texttt{ex:pers1}; ?mbox \rightarrow \text{"}\texttt{karl@ex.com}\text{"}$

- $\mu_3 : ?p \rightarrow \texttt{ex:pers3}; ?name \rightarrow \texttt{"Karl"}$

Die Lösungsabbildungen μ_1 und μ_2 sind kompatibel, während μ_3 weder mit μ_1 noch μ_2 mit kompatibel ist. □

Für diese Arbeit ist auch Gedanke einer Teillösung eines Basis-Graphmusters von Interesse. Eine Lösungsabbildung wird als Teillösung einer anderen bezeichnet, wenn beide miteinander kompatibel sind und der Definitionsbereich der ersten Lösungsabbildung ein Teil des Definitionsbereichs der letzteren ist.

Definition 2.11 (Teillösung einer Lösung)**.** *Seien μ_1 und μ_2 zwei kompatible Lösungsabbildungen eines Basis-Graphmusters bzgl. eines RDF-Graphen. μ_1 ist eine* Teillösung *von μ_2, falls μ_1 und μ_2 kompatibel und* $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ *gilt.*

Die Teillösungsbeziehung zwischen zwei Lösungsabbildungen wird durch die Notation $\mu_1 \subset \mu_2$ ausgedrückt.

2.2.3 Graphmuster

Um in einem RDF-Graph komplexe Strukturen identifizieren und entsprechende Daten extrahieren zu können, können Basis-Graphmuster durch Operatoren miteinander kombiniert werden. Das resultierende Konstrukt wird als *Graphmuster* bezeichnet. Im Folgenden werden drei grundlegenden Möglichkeiten zur Kombination von Basis-Graphmustern eingeführt: Verundung/Join, optionales Graphmuster und Vereinigung.

Zwei Basis-Graphmuster sind miteinander *verundet*, wenn diese durch kein weiteres Schlüsselwort miteinander verknüpft sind. In diesem Fall müssen beide Basis-Graphmuster im RDF-Graphen vorkommen und die Lösungsabbildungen kompatibel sein. Eine Verundung entspricht einer Join-Operation in der relationalen Algebra.

Ein *optionales Graphmuster* wird in einer Anfrage mittels des Schlüsselworts `OPTIONAL` gekennzeichnet. Wenn zwei Basis-Graphmuster durch `OPTIONAL` miteinander verknüpft wurden, so werden nur dann zusätzliche Variablenbindungen von dem optionalen Teil generiert, wenn dieses Basis-Graphmuster in dem RDF-Graph vorkommt. Es ist mit einem Left-Outer-Join in der relationalen Algebra vergleichbar.

Das Schlüsselwort `UNION` kennzeichnet die Vereinigung der Lösungen zweier Basis-Graphmuster. Das heißt, die Menge der Lösungen des einen Basis-Graphmusters werden mit der Menge des Lösungen des anderen zusammengeführt, ohne dass die eine die andere beeinflusst. Die Semantik ist analog zu der Union-Operation in der relationalen Algebra.

Für tiefer gehende Details wird auf die SPARQL-Spezifikation [PS13] verwiesen.

2.2.4 Sternförmige Graphmuster

Das in dieser Arbeit definierte Speichermodell basiert auf der Gruppierung von Tripeln mit demselben Subjekt. Daher sind Basis-Graphmuster deren Tripelmuster dasselbe Subjekt haben von besonderem Interesse.

Definition 2.12 (Sternförmiges Basis-Graphmuster)**.** *Ein* sternförmiges Basis-Graphmuster *(*engl. starlike basic graph pattern*) ist ein Basis-Graphmuster, bei dem alle darin enthaltenen Tripelmuster ein gemeinsames Subjekt (Ressource oder Variable) haben.*

$$p \in \mathrm{BGP} \textit{ ist sternförmig} \Leftrightarrow \forall t_i, t_k \in p \exists s \in \mathbb{I} \cup \mathbb{V} : t_i^s = s \wedge t_k^s = s$$

Abbildung 2.4 zeigt zwei Beispiele für sternförmige Graphmuster. Beim ersten ist das gemeinsame Subjekt eine Variable und beim zweiten eine Ressource.

```
{ ?x rdf:type foaf:Person ;
       foaf:name ?name ;
       foaf:mbox "john.due@example.com . }

{ ex:proc1 rdf:type ex:Proceeding;
       dcterms:editor ?person;
       dcterms:title ?title . }
```

Abbildung 2.4: Zwei Basis-Graphmuster in Sternform

Zur verbesserten Lesbarkeit werden die folgenden Begriffe eingeführt: Im Folgenden wird ein sternförmiges Basis-Graphmuster kurz als sternförmiges Graphmuster oder Sternmuster bezeichnet. Darüber hinaus wird die Phrase *Subjekt des Graphmusters* verwendet, um das gemeinsame Subjekt aller Tripelmuster eines Basis-Graphmusters zu referenzieren. Beispielsweise ist `?x` das Subjekt des ersten Graphmusters von Abbildung 2.4.

Für diese Arbeit sind vor allem die bzgl. desselben Subjekts größtmöglichen sternförmigen Graphmuster einer Anfrage von besonderem Interesse. Daher wird für den Rest der Arbeit vereinbart, dass mit sternförmigem Graphmuster immer die größtmöglichen Sternmuster gemeint sind. Auf Abweichungen wird an den jeweiligen Stellen hingewiesen.

Da ein einzelnes Tripel als eine Spezialform eines Sternmusters betrachtet werden kann, kann offensichtlich jedes Graphmuster in eine Menge von Sternmustern zerlegt werden.

Kapitel 3

Native Verwaltung von RDF-Daten

Seit den Anfängen der Ausarbeitung des Resource Description Frameworks war die Speicherung von RDF-Daten ein Thema. Anfänglich setzten Forscher vor allem auf bereits etablierte, meist relationale Datenbankmanagementsysteme (RDBMS) als Grundlage für ein RDF-Managementsystem und konstruierten Abbildungen des RDF-Datenmodells auf das Datenmodell des gewählten DBMS. Im Laufe der Zeit zeigte sich jedoch, dass das in RDF definierte, graphbasierte Datenmodell nicht besonders effektiv auf das relationale Datenmodell abgebildet werden kann. Als Konsequenz entstanden einige Vorschläge zur nativen Verwaltung von RDF-Daten, d.h. das jeweilige Speichermodell wurde speziell auf das RDF-Datenmodell zugeschnitten.

Wie in Abbildung 3.1 verdeutlicht, ist das Thema dieses Kapitels die physische Speicherung von RDF-Daten und Zugriffsstrukturen auf diese, d.h. Inferenzen spielen eine untergeordnete Rolle. Bezogen auf die Schichtenarchitektur von Datenbanken steht die interne Sicht auf die Daten im Mittelpunkt. Abbildung 3.2 gibt einen Überblick über die Kategorien von RDF-DBMS. Es können grundsätzlich zwei Arten unterschieden werden: (i) DBMS basierte Modelle, die in schemaunabhängige, schemaabhängige und hybride unterteilt werden können (vgl. Abschnitt 3.1.1), und (ii) native, speziell auf RDF zugeschnittene Modelle (vgl. Abschnitt 3.1.2).

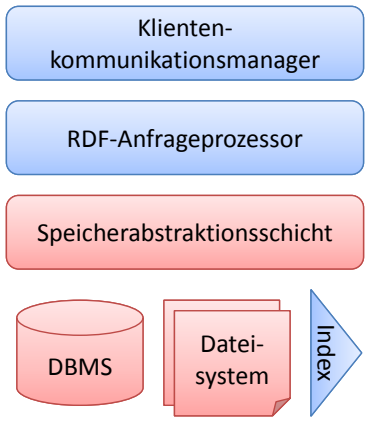


Abbildung 3.1: Generische Architektur eines RDF-Managementsystems

Von den seit der Veröffentlichung des W3C-Spezifikation zum Resource Description Framework angedachten und implementierten Systemen konnten sich jedoch nur wenige in Forschung und Wirtschaft etablieren; zu diesen gehören AllegroGraph, Jena2, Sesame und Virtuoso. Tabelle 3.1 gibt einen Überblick über

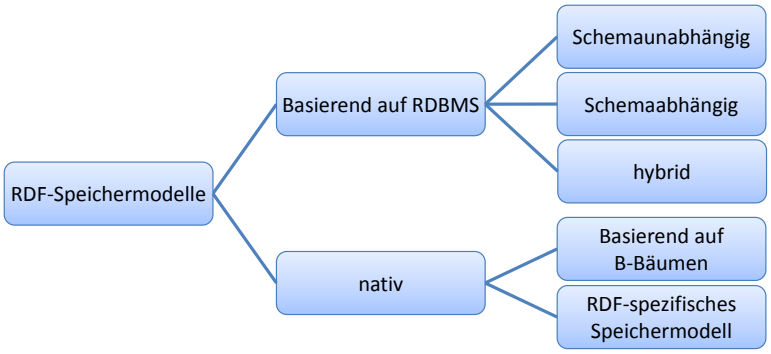


Abbildung 3.2: Kategorisierung der Speichermodelle

existierende RDF-Managementsysteme mit Informationen über das verwendete Speichermodell, den Entwicklungsstatus und weiteren Besonderheiten.

Im Folgenden werden zunächst in Abschnitt 3.1 existierende Ansätze zur Verwaltung von RDF-Daten vorgestellt und diskutiert, wobei sowohl auf relational basierte als auch auf native Speichermodelle eingegangen wird. Anschließend wird in Abschnitt 3.2 ausgehend von den Charakteristiken des RDF-Datenmodells, der Daten und SPARQL-Anfragen ein neuartiges, natives Speichermodell und dessen Operationen präsentiert. In Abschnitt 3.3 erfolgt eine theoretische Analyse der Datenkomplexität und der Komplexität der Operationen. Mögliche Erweiterungen des RCS-Datenmodells werden in Abschnitt 3.4 beschrieben. Die Inhalte und Erkenntnisse aus diesem Kapitel werden abschließend in Abschnitt 3.5 zusammengefasst.

3.1 Existierende Speichermodelle für RDF-Daten

Bevor die Idee der benannten Graphen (*engl. named graphs*) Verbreitung fand, basierten die Speichermodelle auf dem effektiven Verwalten von Tripeln; danach standen Quadrupel im Mittelpunkt der Betrachtungen. Wie sich in den nachfolgenden Abschnitten zeigen wird, werden bei dem existierenden System die Quadrupel jedoch einzeln und nicht im Sinne eines Graphen betrachtet. D.h. die durch den Graphen gegebenen Zusammenhänge zwischen den Ressourcen werden nicht berücksichtigt. Diese Aussage trifft insbesondere auch auf existierende, native Speichermodelle zu. Im Folgenden werden zunächst relational basierte und anschließend native Ansätze betrachtet.

3.1.1 Relational basierte Speichermodelle

Die Basis für die relational basierten Speichermodelle müssen nicht unbedingt relationale DBMS bilden, sondern es können auch spaltenorientierten Systeme (*engl. column store*) verwendet werden [AMH07], die intern anstelle von n-ären Relationen nur mit binären arbeiten. Obwohl diese bei der Verarbeitung von RDF-Daten ein anderes Performanzverhalten zeigen, werden diese im Folgenden aufgrund ihres zum relationalen ähnlichen Datenmodells gemeinsam

System	Literatur	Speichermodell	Status	Besonderheiten
3store	[HS05]	schemaunabhängig	Open Source, inaktiv	
AllegroGraph	[Fra10]	nativ	kommerziell	
gStore	[ZzC+13]	nativ	wiss. Beitrag	basierend auf Graph-DB
Hexa-Store	[WKB08]	Tripelindex	Open Source	
Jena2 SDB	[WSKR03]	hybrid	Open Source	teilweise Normalisierung
Jena2 TDB	[Jen10]	nativ	Open Source	
Kowari	[WGA05]	nativ	Open Source, inaktiv	
OntoQuad	[PPD+13]	nativ	kommerziell	
Oracle	[CDES05]	schemaunabhängig	kommerziell	materialisierte Join-Sichten
RDF-3x	[NW10]	Tripelindex	wiss. Beitrag	
RDFSuite	[ACK+01]	hybrid	Entwicklung eingestellt	Vererbung von Tabellen für Unterklassen
RStar	[MSP+04]	hybrid	wiss. Beitrag	
Sesame	[BKvH02]	schemaabhängig, nativ	Open Source	
System Π	[WLHW09]	nativ	wiss. Beitrag	Index für transitive Eigenschaften
Virtuoso	[Ope10]	nativ	kommerziell	
YARS2	[HUHD07]	nativ	wiss. Beitrag	
N.N.	[MAYU05]	schemaabhängig	wiss. Beitrag	Relation über Pfade

Tabelle 3.1: Übersicht über RDF-Managementsysteme, deren verwendete Speichermodelle und Besonderheiten

mit relationalen DBMS betrachtet. Auf Besonderheiten der spaltenorientierten Systeme wird gegebenenfalls hingewiesen.

Bei der Verwendung eines relationalen DBMS für die Speicherung von RDF-Daten wird eine Abbildung des RDF-Datenmodells auf das relationale vorgenommen. Basierend auf der Klassifikation von [TCK05] lassen sich dabei die Ansätze grundlegend in die folgenden drei Kategorien aufteilen:

Schemaunabhängig (*engl. schema-oblivious*) Das Datenbankschema besteht aus einer einzigen Relation, in der alle Tripel eines RDF-Graphen abgelegt werden.

Schemaabhängig (*engl. schema-aware*) Die Datenbankschemas hängt unmittelbar von den im RDF-Schema vorkommenden Eigenschaften und Konzepten ab, da diese direkt als Tabellen modelliert werden.

Hybrid (*engl. hybrid*) In hybriden Ansätzen werden Teile von schemaunabhängige und schemaabhängige Datenbankschemata miteinander kombiniert, um die Vorteile der beiden Ansätze miteinander zu verknüpfen.

Im verbleibenden Teil dieses Abschnitts werden konkrete Ausprägungen der jeweiligen Kategorien beschrieben und diskutiert. Abschließend werden

generelle Erweiterungen skizziert, die in allen vorgestellten Ansätze Anwendung finden können.

3.1.1.1 Schemaunabhängig

In dem einfachsten Ansatz werden die RDF-Daten unabhängig von den vorkommenden Ressourcen und Eigenschaften in einer einzigen Relation *R(Subjekt, Prädikat, Objekt)* verwaltet. Dabei entspricht ein Eintrag in der Tabelle genau einer Aussage im RDF-Graph. In [OBW06] wird zwar ein derartiges System beschrieben, jedoch werden an den meisten schemaunabhängigen Datenbankschemata Modifikationen vorgenommen, um eine effizientere Verarbeitung von Anfragen zu erreichen (siehe zum Beispiel unten *Normalisierung von IRIs und Literalen*).

Bei der Verwendung von normalisierten IRIs und Literalen wird es erforderlich, den Typ eines Objektwertes (Ressource oder Literal) abzuspeichern, damit das Ergebnis einer Anfrage keine IDs sondern die eigentlichen Werte enthält. Im Zuge dessen entstanden Variationen der obigen Relation [WSKR03, HBS08]: (i) *R(Subjekt, Prädikat, ObjektURI, ObjektLiteral)* und (ii) *R(Subjekt, Prädikat, isLiteral, Objekt)*. In der ersten Variation werden Objekt-Ressourcen und Objekt-Literalen in verschiedenen Attributen der Relation abgelegt, wobei eines der beiden Attribute immer `null` ist (vgl. Abbildung 3.3). Des Weiteren ließe sich dieses Datenbankschema noch dahingehend verfeinern, dass Literale zusätzlich entsprechend ihres XML-Schema-Datentyps [BM04] gespeichert werden. Das heißt, anstelle des Attributs *ObjektLiteral* der Tripelrelation tritt eine Menge von Attributen, die den Datentypen in XML-Schema entsprechen [DS09]. Die zweite Variation nutzt lediglich einen booleschen Wert als Diskriminator.

`Tripel`

Subjekt	Eigenschaft	ObjektURI	ObjektLiteral
`ex:pers1`	`rdf:type`	`foaf:Person`	
`ex:pers1`	`foaf:name`		`"John Due"`
`ex:proc1`	`rdf:type`	`ex:Proceeding`	
`ex:proc1`	`dcterms:editor`	`ex:pers1`	

Abbildung 3.3: Tripelrelation

Ein wesentlicher Vorteil des schemaunabhängigen Ansatzes liegt in dem geringen Aufwand beim Importieren von RDF-Daten, da keine Änderungen am Datenbankschema während das Importvorgangs vorgenommen werden müssen. Dem gegenüber stehen die Nachteile, dass für jede Anfrage die gesamte Datenbasis betrachtet werden muss und dass die Lokalität von Subjekten und Eigenschaften nicht ausgenutzt werden kann [WSKR03]. Gerade bei der Bearbeitung von Anfragen mit vielen Tripelmustern oder einer geringen Selektivität von Tripelmustern kann dies zu Performanzeinbußen führen: Entweder werden viele Self-Join-Operationen benötigt oder die Datentabelle muss beim inkrementellen Generieren von Ergebnissen oft zugegriffen werden. Abbildung 3.4 zeigt ein Beispiel für eine Anfrage an ein schemaunabhängiges Speichermodell, die nach den Namen von Editoren fragt. Bei der Verwendung

eines normalisierten Schemas werden zusätzlich noch Join-Operationen benötigt, um die normalisierten Werte aufzulösen.

```
SELECT t1.ObjektLiteral
FROM Tripel t1, Tripel t2
WHERE
        t1.Subjekt = t2.Subjekt AND
        t1.Eigenschaft = 'foaf:name' AND
        t2.ObjektURI = 'dcterms:editor'
```

Abbildung 3.4: SQL-Anfrage an ein schemaunabhängiges Speichermodell

Ein komplexerer, aber dennoch schemaunabhängiger Ansatz bildet eine vorher festgelegte Menge an Eigenschaften auf separate Relationen ab. Hierfür werden Eigenschaften ausgewählt, die in grundlegenden Spezifikationen wie RDF/S oder OWL definiert sind und häufig Verwendung finden, z. B. `rdfs:subClassOf` und `rdf:type`. [GSV04, Bro05, MAYU05]

3.1.1.2 Schemaabhängig

In einem schemaabhängigen Ansatz hängt das Datenbankschema von den in den RDF-Daten vorkommenden Eigenschaften und/oder Klassen ab. Dabei können drei Arten von Relationen unterschieden werden: einfache Eigenschaftsrelationen (*engl. property table*), gruppierte Eigenschaftsrelationen (*engl. clustered property table*) und Eigenschaft-Klassen-Relationen (*engl. property-class table*). [WSKR03]

Einfache Eigenschaftsrelationen sind binäre Relationen $Property_i$*(Subjekt, Objekt)* und zeichnen sich dadurch aus, dass diese nur die Instanzen einer Eigenschaft $Property_i$ enthalten (vgl. Abbildung 3.5). Bestünde das Speichermodell nur aus Eigenschaftsrelationen, so würde das Schema letztendlich so viele Relationen enthalten, wie es Eigenschaften in den RDF-Daten gibt. Ein großer Nachteil von Eigenschaftsrelationen ergibt sich, wenn Anfragen mit ungebundenen Prädikaten gestellt werden, da erst zur Laufzeit die anzufragenden Tabellen bestimmt werden müssen.

`rdf_type`

Subjekt	Objekt
ex:pers1	foaf:Person
ex:pers1	ex:Proceeding

`foaf_name`

Subjekt	Objekt
ex:pers1	"John Due"

`dcterms_editor`

Subjekt	Objekt
ex:proc1	ex:pers1

Abbildung 3.5: Eigenschaftsrelationen

Das Ziel von gruppierten Eigenschaftsrelationen besteht im effizienten Zugreifen auf oft gemeinsam angefragte Eigenschaften einer Ressource. Dabei werden die Eigenschaften als Attribute einer Relation *R(Subjekt*, $Property_1$, ..., $Property_n$*)* modelliert (vgl. Abbildung 3.6). Insofern die Datenbank keine mehrwertigen Attribute unterstützt, eignet sich diese Art von Eigenschaftsrelationen nur für einwertige Eigenschaften. Eigenschaft-Klassen-Relationen

sind spezielle gruppierende Eigenschaftsrelationen, die sich auf die Instanzen einer konkreten RDF-Klasse beschränken.

Person

Subjekt	rdf_type	foaf_name
ex:pers1	foaf:Person	"John Due"

Proceeding

Subjekt	rdf_type	dcterms_editor
ex:proc1	ex:Proceeding	ex:pers1

Abbildung 3.6: Gruppierte Eigenschaftsrelation

Als Ergänzung zu den Eigenschaftsrelationen können schemaabhängige Speichermodelle separate Tabellen für spezielle Informationen enthalten, z. B. eine Relation aller RDF-Klassen oder aller Eigenschaften oder spezielle Strukturen für Reifikation. Darüber hinaus kann einem Attribut einer Eigenschaftsrelation ein geeigneter SQL-Datentyp zugewiesen werden, wenn der Wertebereichsdatentyp einer Eigenschaft bekannt ist. Dadurch werden einer Datenbank weitergehende Möglichkeiten zur Speicherung und Optimierung von Anfragen eröffnet.

Einige objektrelationale Datenbanken erlauben das Erstellen von Relationen, die zueinander in einer Vererbungshierarchie stehen. In [BKvH02] wurde anhand von PostgreSQL[1] untersucht, inwiefern sich diese Funktionalität zur Abbildung der Unterklassenbeziehung zwischen Ressourcen eignet. Letztendlich beurteilten Broekstra et al. die Resultate als unbefriedigend, da insbesondere das Einfügen einer Klasse in eine existierende Unterklassenbeziehung einen kompletten Neuaufbau aller Relationen in dieser Hierarchie erforderte.

Der Vorteil der schemaabhängigen Ansätze liegt aufgrund der Verteilung der Daten auf mehrere Relationen in der Vermeidung von Self-Joins. Insbesondere Anfragen, bei denen nur eine Relation zum Berechnen des Ergebnisses benötigt wird, können effizient beantwortet werden (einfache SELECT-Anfrage). Im Gegensatz dazu werden jedoch bei komplexeren Anfragen, in denen mehrere Relationen involviert sind, Join- und Union-Operationen zum Zusammenführen der Daten benötigt (vgl. Abbildung 3.7). [AMH07] Falls beispielsweise in einem Tripelmuster einer Anfrage eine Eigenschaft beliebig sein kann („don't care"), müssen alle Relationen durchsucht werden. Gleiches gilt auch, wenn die Klasse nicht spezifiziert worden ist.

```
SELECT n.Objekt
FROM foaf_name n, dcterms_editor e
WHERE n.Subjekt = dcterms_editor.Subjekt

SELECT foaf_name
FROM Person, Proceeding
WHERE Person.Subjekt = dcterms_editor
```

Abbildung 3.7: SQL-Anfrage an das schemaabhängige Speichermodell aus den Abbildungen 3.5 (oben) und 3.6 (unten)

[1] http://www.postgresql.org/

Wenn außerdem nur wenige Ressourcen für dieselben Eigenschaften Werte besitzen – die RDF-Daten sind eher unstrukturiert –, so entstehen spärlich besetzte Eigenschaftsrelationen. Die damit einhergehenden nu11-Werte in den Relationen können zu weiteren Performanzeinbußen beitragen [ASX01, BHKN06].

Weiterhin können Änderungen der Daten, insbesondere das Einfügen neuer Tripel, Auswirkungen auf das Schema haben. Wird beispielsweise ein Tripel hinzugefügt, wodurch ein Attribut einer gruppierten Eigenschaftsrelation mehrwertig wird, sind Schemaanpassungen erforderlich: Das betroffene Attribut muss aus der Relation entfernt und eine einfache Eigenschaftsrelation für dieses erzeugt werden.

3.1.1.3 Hybrid

Aufgrund der genannten Nachteile der beiden zuvor vorgestellten Ansätze wurden diese nur selten in ihrer reinen Form realisiert. Stattdessen wurden schemaunabhängigen und schemaabhängige zu den sogenannten hybriden Speichermodellen kombiniert. Der Grundgedanke ist dabei, Konzepte und Eigenschaften nach Möglichkeit auf separate Relationen aufzuteilen und nur die übrig bleibenden Aussagen in einer Tripelrelation abzuspeichern. Die Grundlage für das Aufteilen auf Relationen können dabei Gemeinsamkeiten von Aussagen oder die Häufigkeit einer Eigenschaft in den RDF-Daten oder in Anfragen bilden.

In [TCK05] wird beispielsweise ein Datenbankschema beschrieben, in dem eine binäre Relation für alle Klasseninstanzen existiert. Entsprechend dem schemaabhängigen Speichermodell werden anhand des Datentyps ihres Objekts die Aussage unterschiedlichen Relationen zugeordnet, z. B. $R_1(S,P,O_{Res})$, $R_1(S,P,O_{int})$,..., wobei O_{Res} Ressource und O_{int} ganzzahlige Literale bezeichnen. Die Tatsache jedoch, dass für die Zuordnung nur der Datentyp des Objekts aber nicht die Eigenschaft selbst ausschlaggebend ist, ist charakteristisch für ein schemaunabhängiges Schema.

Ein anderer Ansatz wird in [HBS08], [Bro05] und [Kim05] skizziert, bei dem Aussagen mit vorher festgelegten Eigenschaften in separaten Relationen $P_i(S,O)$ verwaltet werden. Dies können Eigenschaften aus RDF-relevanten Spezifikationen/Standards oder aber auch andere häufig in den Daten auftretende bzw. häufig angefragte Eigenschaften sein. Hertel betrachtet beispielsweise die nachfolgend aufgelisteten RDFS-Eigenschaften als Relationen, da mit diesen das Ontologie-Schema von den Instanzen getrennt wird. Des Weiteren werden Klassen- und Eigenschaftshierarchien explizit gespeichert.

- Unterklasse – `rdfs:subClassOf`
- Untereigenschaft – `rdfs:subPropertyOf`
- Definitionsbereich – `rdfs:domain`
- Wertebereich – `rdfs:range`
- Klassen-Instanzen – `rdfs:type` mit einer Klasse als Objekt
- Eigenschaft-Instanzen – `rdfs:type` mit einer Eigenschaft als Objekt

In Jena Version 2 (kurz Jena2) [WSKR03] wurde ein hybrider Ansatz realisiert, bei dem es neben einer Tripelrelation auch noch gruppierte Eigenschaftsrelationen $R(S, P_1, \ldots, P_n)$ von häufig gemeinsam angefragten Eigenschaften gibt (siehe Beispiel in Abbildung 3.6). Die Vor- und Nachteile von Eigenschaftsrelationen wurden schon in Abschnitt 3.1.1.2 auf Seite 19 beschrieben. In Kombination mit einer generischen Tripelrelation lassen sich jedoch im Vergleich zu einem auf Eigenschaftsrelationen basierendem Schema schwach besetzte Tabellen besser vermeiden und des Weiteren können mehrwertige Eigenschaften verwaltet werden.

Die genannten Ansätze übertragen Teile des den RDF-Daten zugrunde liegenden Ontologieschemas explizit auf das Datenbankschema und trennen es von den Instanzdaten. Sie versuchen somit einen Kompromiss zwischen geringem Aufwand für die notwendigen Anpassungen des Datenbankschemas bei Veränderungsoperationen (z. B. Einfügen und Löschen von Tripeln) und Anfrageperformanz zu erzielen. In [TCK05] wird darüber hinaus das Beibehalten der Typinformationen in der Datenbank als Vorteil genannt. Abbildung 3.8 zeigt einen Ausschnitt aus einem hybriden Datenbankschema. Die Relationen zur Abbildung von wichtigen Eigenschaftsbeziehungen werden analog zur Relation `rdf_type` definiert. Beispielsweise wurden die Eigenschaften `foaf_name` und `foaf_img` zusammen in einer Relation gruppiert.

`rdf_type`

Subjekt	ObjektURI
`ex:pers1`	`foaf:Person`

`Person`

Subjekt	foaf_name	foaf_img
`ex:pers1`	`"John Due"`	`http://.../due.png`

`Tripel`

Subjekt	Eigenschaft	ObjektURI	ObjektLiteral
`ex:proc1`	`dcterms:editor`	`ex:pers1`	

Abbildung 3.8: Hybrides Speichermodell mit einer Relation für den Ressource-Typ, gruppierter Eigenschaftsrelation für Person und einer Tripelrelation für die restlichen Tripel (Ausschnitt)

3.1.1.4 Allgemeine Erweiterungen und Variationen

Unabhängig von dem verwendeten Speichermodell lassen sich weitere Modifikationen vornehmen, die vor allem auf eine Verbesserung des Anfrageverhaltens abzielen. In diesem Abschnitt wird auf Normalisierung von IRIs und Literalen, die Verwendung von materialisierten Sichten sowie die Speicherung von benannten Graphen eingegangen.

Normalisierung Unabhängig von der Art des Speichermodells benötigt die Speicherung von IRIs aufgrund ihrer Länge und ihrem Auftreten in vielen Tripeln größeren Speicherplatzbedarf. Ähnliches trifft auch auf Literale zu, wenn dieselben Literale häufig in den RDF-Daten auftreten. Daher wurden zunächst sogenannte normalisierte Schemata (*engl. normalized schemtas*) entwickelt, in

denen alle IRIs und Literale durch nummerische [CDES05] oder auf Hashwerten basierende Identifikatoren ersetzt werden [HG03]. Die Abbildungen zwischen IRI und ID bzw. Literal und ID werden entweder in einer einzelnen Tabelle oder in zwei separaten Tabellen gespeichert. Als Konsequenz der Normalisierung sind jedoch zusätzliche Join-Operationen erforderlich, um das Ergebnis einer Anfrage zu generieren.

Bei der Evaluation der Performanz ihrer RDF-Managementsysteme kamen Oldakowski et al. (RAP) [OBW06] und Wilkinson (Jena Version 1) [WSKR03] zu dem Schluss, dass die Normalisierung ein Performanzproblem darstellt. Erstere berichten für ein komplett denormalisiertes Speichermodell von einer Beschleunigung der Anfragebearbeitung um das Zwei- bis Dreifache – den Speicherplatzbedarf bezeichnen sie dabei als akzeptabel, weil sie mit RAP nur auf die Verwaltung von „mittelgroßen“ RDF-Graphen abzielen. Letzterer hingegen suchte nach einer Lösung, die einen Kompromiss zwischen Zeit- und Speichereffizienz darstellt. In Jena2 wurde ein partiell normalisiertes Schema realisiert, in dem zwei Verfahren zur Kompression umgesetzt sind:

1. Gemeinsame IRI-Präfixe wie Namensräume werden durch einen kürzeren Wert ersetzt. Hierbei gehen die Autoren davon aus, dass die Anzahl der zu ersetzenden Präfixe klein ist und die Abbildungstabelle im Hauptspeicher gehalten werden kann (vgl. Abbildung 3.9 – (1)).

2. Ersetzen von Literalen und IRIs durch einen nummerischen Identifikator, wenn deren Länge einen vorher festgelegten Grenzwert überschreiten (vgl. Abbildung 3.9 – (2) und (3)).

(1) `http://www.w3.org/1999/02/22-rdf-syntax-ns#Property` ⇔ `rdf:Property`

(2) `"Literal, länger als der Grenzwert."` ⇔ `101`

(3) `ex:RessourceMitLangemLokalemNamen` ⇔ `102`

Abbildung 3.9: Kompression von IRIs und Literalen

Materialisierte Sichten In heutigen DBMS werden materialisierte Sichten eingesetzt, um komplexe Anfragen zu Lasten des Speicherplatzes zeiteffizient beantworten zu können. Mit dem Begriff materialisierte Sicht bezieht man sich auf das Speichern des Ergebnisses einer Anfrage in der Datenbank [Dat06] – manchmal wird auch der Terminus *Vorberechnung* in diesem Zusammenhang verwendet.

Da in SPARQL-Anfragen auch komplexere Graphstrukturen in den RDF-Daten gesucht werden können, finden materialisierte Sichten auch in RDF-Managementsystemen Anwendung. Die Autoren in [CDES05] verwenden materialisierte Join-Sichten, um kostenintensive Mehrwege-Joins bei der Auswertung von Anfragen mit mehreren Tripelmustern zu vermeiden. Aus den sechs möglichen Zweiwege-Joins auf einer Tripeltabelle $S \bowtie S$, $S \bowtie O$, $O \bowtie O$, $S \bowtie P$, $P \bowtie P$, und $P \bowtie O$ wurden nur die ersten drei als besonders nützlich eingestuft. Darüber hinaus werden materialisierte Sichten für häufig gemeinsam

auftretende, einwertige Eigenschaften erzeugt – es entstehen also gruppierte Eigenschaftsrelationen als Sichten auf die Tripelrelation.

Benannte Graphen Die in den vorherigen Abschnitten beschriebenen Ansätze lassen sich ohne großen Aufwand auf die Verwaltung von benannten Graphen erweitern. Mit Hilfe des Konzepts von benannten Graphen [CBHS05] wird eine Menge von Aussagen zusammengefasst und mit einer IRI *benannt*. Je nach verwendetem Speichermodell wird den Relationen entweder ein weiteres Attribut G hinzugefügt. Diese enthält die IRI des Graphen, zu der eine Aussage gehört (z. B. $R(G, S, P, 0)$. Alternativ wird die Bezeichnung der Relationen so gewählt, dass sie Rückschlüsse auf den zugehörigen benannten Graphen zulässt (z. B. eine Relation mit der Bezeichnung R_G würde Aussagen des benannten Graphen G beinhalten). Da Graphen auch durch eine IRI identifiziert werden, ist hierbei eine Normalisierung sinnvoll.

3.1.2 Tripelindex basierte Speichermodelle

Für einen beschleunigten Zugriff auf die RDF-Daten von RDBMS-basierte Speichermodellen wurden oftmals Indexe über Kombinationen von Subjekt, Prädikat und Objekt angelegt. Da diese Indexe letztendlich die kompletten RDF-Daten widerspiegelten, wurde im weiteren Verlauf auf das Erstellen einer Tripeltabelle verzichtet und Systeme entwickelt, die ausschließlich aus den Indexen, üblicherweise B+-Bäumen, bestehen.

In [HD05] wurde untersucht, welche Indexe minimal für ein effizientes Anfragen von RDF-Daten geeignet sind. Betrachtet man SPARQL-Anfragen, so können mit einem Tripelmuster acht und mit einem Quadrupelmuster 16 verschiedene Zugriffsmuster generiert werden – jede Komponente kann eine Variable oder eine Konstante sein. Ausgehend von den möglichen Zugriffsmustern wurden sechs kombinierte Indexe abgeleitet, mit denen sich alle 16 Zugriffsmuster abgedeckten lassen (vgl. Abbildung 3.10). Ein auf diesen sechs Indexen basierendes Speichermodell wurde dann später in dem System YARS2 realisiert [HUHD07].

SPOG	POG	OGS	GSP	GP	OS
(?:?:?:?)	(?:p:?:?)	(?:?:o:?)	(?:?:?:g)	(?:p:?:g)	(s:?:o:?)
(s:?:?:?)	(?:p:o:?)	(?:?:o:g)	(s:?:?:g)		
(s:p:?:?)	(?:p:o:g)	(s:?:o:g)	(s:p:?:g)		
(s:p:o:?)					
(s:p:o:g)					

Abbildung 3.10: Sechs kombinierte Indexe überdecken die 16 möglichen Zugriffsmuster auf Quadrupel (? = Variablen; s, p, o, g = Konstanten)

Werden benannte Graphen bei der Speicherung nicht berücksichtigt, so kann die Anzahl der benötigten Indexe auf drei reduziert werden (SPO, POS und OSP). Dieser Ansatz wurde in den Systemen Jena TDB [Jen10], AllegroGraph [Fra10], Virtuoso [Ope10], System Π [WLHW09] und Kowari [WGA05] umgesetzt. Während Jena TDB, System *Pi* wie auch YARS2 auf B+-Bäume

setzt, werden in AllegroGraph und Kowari anstelle derer binäre Bäume verwendet. Die Indexe in Jena TDB wurde sogar noch speziell auf den Anwendungsfall zugeschnitten. Vituoso hingegen verwendet extensiv Bitmap-Indexe, um einen effizienten Zugriff zu gewährleisten[2], und Kompression, um den benötigten Speicherplatz zu reduzieren. Darüber hinaus unterstützen Jena TDB, AllegroGraph und Virtuoso auch benannte Graphen, indem ausgehend von den genannten Tripelindexen zusätzliche Indexe für Quadrupel erzeugt werden ($xxxG$ und $Gxxx$). Die Besonderheit des System Π ist ein zusätzlicher Index über transitive Eigenschaften, um das Inferieren der sich aus der Transitivität ergebenen Tripel zu beschleunigen.

Nur Tripel betrachtend argumentieren Weiss et al. in [WKB08], dass drei Indexe nicht ausreichend seien, um komplexere Anfragen beantworten zu können. Stattdessen präsentieren Sie das Hexastore-System das auf sechs Indexen basiert und damit allen Komponenten eines Tripels in der Anfragebearbeitung die gleiche Bedeutung zumessen. Im Vergleich zu [WGA05] werden daher drei zusätzliche Indexe erzeugt, die aus dem Vertauschen der jeweiligen zweiten und dritte Komponente entstehen (SOP, PSO, OPS). Bei kleineren Anfragemustern können somit die Kosten für Join-Operationen reduziert werden. Damit der Speicherplatz im Hexastore nicht das Sechsfache im Vergleich zu einer Tripeltabelle benötigt, werden zum einen die Daten normalisiert und eine spezielle Indexstruktur verwendet. Diese Struktur ermöglicht das gemeinsame Nutzen von Einträgen in zwei Indexen und reduziert den Speicherbedarf auf das Fünffache, zum Beispiel greifen die Indexe SPO und PSO auf dieselben Objektdaten zu. Als weiteren Nachteil könnte der Aufwand angesehen werden, der durch das Aktualisieren der Indexstrukturen bei Änderungsoperationen entsteht. In ihren Veröffentlichungen treffen Weiss et al. jedoch hierzu keine Aussage.

Die Arbeit von Potocki et al. [PPD+13] greifen die Idee des Hexa-Store auf und erweitern den Ansatz um die Fähigkeit, Quadrupel verwalten zu können. Das Speichermodell basiert auf dem vertikalen Partitionieren der RDF-Daten und organisiert diese auf vier Ebenen. Jede Ebene entspricht einer Komponente eines Quadrupels, wobei die nächst tiefere Ebene eine Verfeinerung darstellt. Wenn beispielsweise auf der ersten Ebene eine konkretes Prädikat als Schlüssel gewählt wurde, dann finden sich auf der nächsten Ebene die Subjekte, Objekte und Graphen, die mit diesem Prädikat in einem Tripel vorkommen. Betrachtet man also einen Pfad von der ersten Ebene bis zur vierten Ebene erhält man ein in den zugrunde liegenden RDF-Daten enthaltenes Quadrupel. Auch bei diesem Ansatz wurden in der Veröffentlichung keine Aussagen über den Aufwand von Änderungsoperationen getroffen.

Die Grundlage des Systems RDF-3X von Neumann et al. [NW10] bilden wie beim Hexastore sechs Indexe über die verschiedenen Kombinationen von Subjekt, Prädikat und Objekt. Die beiden Systeme unterscheiden sich jedoch in der Verwaltung der Tripel. In RDF-3X werden komprimierte B+-Bäume eingesetzt, in denen die Tripel in lexikographischer Ordnung abgelegt werden. Die Ordnung der Tripel beschleunigt das Finden von Ergebnissen für ein Tripelmuster, da diese Operation als Bereichsanfrage auf einem Tripelindex durchgeführt werden kann. Eine Kompression der B+-Bäume, die neben der Normalisierung von IRIs und Literalen auf dem Speichern nur des Deltas zwischen

[2] http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSArticleWebScaleRDF

zwei Tripeln beruht, wirkt einer Vervielfachung des benötigten Speicherplatzes entgegen.

Einen Schritt weiter geht [FB09] mit dem Dreiwege-Tripel Baum, in dem alle *Atom* des RDF-Datenmodell (Ressourcen und Literale) unabhängig von ihrer Rolle in den RDF-Daten gleichwertig indexiert werden. Zu jedem Atom werden alle Tripel gespeichert, in denen es als Subjekt, Prädikat oder Objekt auftritt, wobei diese Listen nach der jeweiligen Position gruppiert sind: OP bei Auftreten des Atoms als Subjekt, SO bei Auftreten als Prädikat und SP bei Auftreten als Objekt. Dadurch erreichen die Autoren eine erhebliche Reduktion des Speicherbedarfs für den Index bei guter Performanz.

Der Vollständigkeit halber seien an dieser Stelle noch die Ansätze erwähnt, in denen kein neues Speichermodell sondern nur eine Abbildung zwischen der RDF-Anfragesprache SPARQL und der relationalen Anfragesprache SQL konstruiert wird [ECTOO09, CLF09, Cyg05, MWL+08]. Somit können die Daten wie bisher in relationalen DBMS verwaltet werden und dennoch Semantic-Web-Anwendungen darauf zugreifen.

3.1.3 Weitere Speichermodelle

In diesem Abschnitt werden Ansätze zum Verwalten von RDF-Daten vorgestellt, die nicht in die obigen Abschnitte einsortiert werden konnten oder besondere Aspekte des RDF-Datenmodells im Speichermodell berücksichtigen.

Zou et al. beschreiben in [ZzC+13], wie eine Graph-Datenbank zum Verwalten von RDF-Daten und zum Beantworten von SPARQL-Anfragen mit Platzhaltern und Aggregationen herangezogen werden kann. In dem Speichermodell wird ein RDF-Graph in Form von Adjazentslisten enthaltenden Tabelle gespeichert. Die Adjazentsliste beinhaltet für einen Knoten des RDF-Graphen die ausgehenden Prädikate und Objekte. Dabei wird das Prädikat und das Objekt als Bitset kodiert, die sogenannte Knotensignatur (*engl. vertex signature*). Zum Indexieren der Knotensignaturen wird ein balancierter Baum (VS*-Baum) verwendet. Eine Anfrage wird in gleicher Weise als Bitset kodiert (*engl. query signature graph*). Für jeden Knoten in der Anfragesignatur werden über die Indexstruktur alle Knoten mit derselben Signatur gesucht und anschließend miteinander verknüpft (Join). Abschließend muss noch überprüft werden, ob ein gefundener Kandidat tatsächlich eine Lösung der Anfrage darstellt.

In [WLS03] untersuchten die Autoren inwiefern sich Ontologien in logische Programme transformieren und in deduktive Datenbanksysteme importieren lassen. In ihrem Beitrag wird vor allem die Transformation von OWL-Konstrukte auf Description Logic Ausdrücke beschrieben, die für diese Arbeit nicht relevant ist. Nach der Transformation einer Ontologie wurde das logische Programm in das CORAL-System [RSSS94] geladen. Eine Evaluation dieses Ansatzes zeigte, dass in CORAL Indexierungsmechnismen wie B-Bäume fehlen, um eine performante Anfragezeit zu erreichen. Dieser Ansatz wurde von den Autoren nicht weiter verfolgt.

Das in [MAYU05] beschriebene Speichermodell ist grundlegend zu den schemaabhängigen gehörig anzusehen. Jedoch wurde es um einen Index für Pfade ergänzt, der mit einem Nummerierungsschema arbeitet, wie es von XML-Datenbanken her bekannt ist. In dem Index werden umgekehrte Pfadausdrücke (*engl. reverse path expressions*) von einem aktuellen Knoten hin zu

allen erreichbaren Wurzeln in Form einer Datenbankrelation verwaltet. Die Pfade werden dabei als eine Zeichenkette mit proprietärer Syntax beschrieben. Mit dem Pfadindex sollen dabei nicht der gesamte RDF-Graph sondern nur häufig angefragte Pfadausdrücke abgedeckt werden. Unter Verwendung des Pfadindexes lassen sich somit Anfragen zeiteffizient beantworten, die einen Pfad mit Variablen an dessen Anfang und Ende enthalten.

3.1.4 Erkenntnisse

Obwohl die vorgestellten Speichermodelle teilweise auf recht unterschiedlichen Zugriffsstrukturen basieren, lassen sich doch gemeinsame, grundlegende Ziele bei den Herangehensweisen beobachten; es haben sich im Laufe der Zeit „best practices“ herausgebildet. Die folgende Aufzählung listet wesentliche Ziel von RDF-Managementsystem auf:

Reduktion von Join-Operationen Aufgrund des einfachen, auf Tripeln basierenden RDF-Datenmodells stellt das Verknüpfen von diesen eine wesentliche und häufig verwendete Operation dar, die im Allgemeinen die größten Kosten bei der Berechnung von Anfrageergebnissen verursacht. Eine Verringerung der Anzahl von Join-Operationen wirkt sich demzufolge unmittelbar auf die Antwortzeiten eines Systems aus.

Normalisierung Die Verwaltung der unter Umständen sehr langen Zeichenketten, die durch IRIs und insbesondere Literale entstehen, erfordert besondere Berücksichtigung in einem RDF-Managementsystem. Zum einen verursachen Vergleichsoperationen auf diesen während der Anfragebearbeitung höhere Kosten und zum anderen benötigen sie bei häufigerem Auftreten mehr Speicherplatz. Aus diesen Gründen verbessert eine Substitution der Zeichenketten durch kürzere Identifikatoren die Performanz eines Systems hinsichtlich der Anfragebearbeitungszeit und des Speicherplatzbedarfs.

Geringer Speicherplatzbedarf Neben IRIs und Literalen benötigen auch die Zugriffsstrukturen für das Lokalisieren von Teilgraphen zusätzlichen Speicherplatz. Trotz der heutzutage günstigen Hardware wird ein moderater Umgang angestrebt, um eine Vervielfachung der RDF-Daten aufgrund von Indexierung und somit Aufwände bei Operationen auf den Daten (Importieren in den/Laden vom Sekundärspeicher, Datenaktualisierung) zu verringern. An dieser Stelle muss ein Kompromiss zwischen Speicherplatz und Anfrageperformanz eingegangen werden.

Gleichgewichtung Die Zugriffsstrukturen sind idealerweise so konstruiert, dass sie für beliebige RDF-Graphen geeignet sind und somit keine besonderen Annahmen über deren Struktur treffen. Beispielsweise werden Subjekt, Prädikat und Objekt gleich gewichtet und der Aufwand für die physische Optimierung des Systems durch den Nutzer eines RDF-Managementsystems wird reduziert.

Mehrwertige, optionale Eigenschaften Auf der Ebene der RDF-Semantik [HPS14] sind für eine Ressource zum einen mehrwertige Eigenschaften gestattet und zum anderen ist das Vorhandensein einer Eigenschaft nicht garantiert. Dies wirkt sich unmittelbar auf das Speichermodell aus.

3.2 Konzeption des Resource Centered Store

Im vorherigen Abschnitt wurden die gängigsten Speichermodelle für RDF-Daten vorgestellt. Neben dem auf extensive Nutzung von materialisierten Sichten beruhenden System von Oracle [Ora14], werden vor allem die auf Tripelindexe aufbauenden, nativen Speichermodelle als besonders performant angesehen. Jedoch bleibt bei letzteren Systemen die Schwachstelle, dass die Berechnung von Anfragen mit vielen Tripelmustern in ebenso häufigen Join-Operationen zwischen Tripelindexen resultiert.

In diesem Abschnitt wird ein neuartiges, natives Speichermodell für RDF-Daten konzipiert, dass vor allem auf das Reduzieren von Join-Operationen abzielt. Dabei setzt das Konzept an der für Datenbanksysteme üblichen physischen Datenverwaltung auf Datenbankseiten an. Anstelle der in RDBMS verwendeten Tupel mit fest definierten Attributen werden unabhängig von den einer Ressource zugeordneten Eigenschaften Teilgraphen eines RDF-Graphen auf einer Datenbankseiten verwaltet.

3.2.1 Überblick

Über die grundlegenden Unterschiede zwischen dem relationalen Datenmodell (Relationen mit Attributen und Werten) dem RDF-Datenmodell (Graphen mit Ressourcen und Beziehungen zwischen diesen) hinaus, werden durch RDF weitere Eigenschaften impliziert, die für das Konzept des Speichermodells für RDF-Daten ausgenutzt werden können. Des Weiteren fließen auch Eigenschaften der RDF-Daten selbst und von typischen Anfragen mit in das Konzept ein. Vor allem wurden im Speichermodell die Eindeutigkeit von IRIs im RDF-Graphen, die Zuordnung vieler Eigenschaften zu einer Ressource sowie die Lokalität von Anfragen ausgenutzt.

Die eindeutige Identifikation von Ressourcen durch IRIs ist eine wesentliche Eigenschaft des RDF-Datenmodells. Betrachtet man einen RDF-Graphen, so kann es in diesem maximal nur einen einzigen Knoten mit einer gegebenen IRI geben.

Das Resource Description Framework wurde vor allem dazu entworfen, Konzepte in einer Anwendungsdomäne (Ressourcen) näher mit Eigenschaften zu beschreiben. Daher ist davon auszugehen, dass beim Beschreiben einer Ressource im Allgemeinen mehrere Eigenschaften zugeordnet werden. Betrachtet man beispielsweise den bislang bekanntesten Datensatz im Bereich Semantic Web, DBpedia, so kann man der Tabelle 2 in [BLK+09] entnehmen, dass das Verhältnis zwischen der Anzahl an Ressourcen und der Anzahl an Tripeln ca. $2,8 \cdot 10^6 : 70,2 \cdot 10^6 = 1 : 25$ beträgt. Das heißt, für jede Ressource sind durchschnittlich 25 Tripel vorhanden. Des Weiteren wird diese Annahme auch durch die anerkannten RDF-Benchmarks SP^2Bench [SHLP09] und BSBM [BS09] gestützt, bei deren sich die Autoren an reale Daten und Szenarien orientierten:

an dem Literaturverzeichnis DBLP[3] bzw. der Modellierung der Anwendungsdomäne Online-Shop.

Jeder Ressource ist aufgrund der RDF-Semantik [HPS14] zumindest der Typ (`rdf:type`) `rdf:Resource` zugeordnet. Jedoch ist oftmals ein konkreterer Typ als dieser gegeben, entweder durch eine explizite Aussage über den Typ einer Ressource in den RDF-Daten oder durch Ableiten dieser Information aus den Angaben von Definitions- und Wertebereich bei Eigenschaften.

Nicht zuletzt kann die Struktur von Anfragen eine bestimmte Anordnung der Daten motivieren. Eine solche Eigenschaft ist die Lokalität von Anfragen. Das heißt, insofern eine Anfrage aus großen Graphmustern besteht, so sind häufig Teile der Anfrage sternförmig angeordnet und miteinander durch Pfade verknüpft sind. Diese Struktur liegt darin begründet, dass zu einer Ressource oft mehrere Eigenschaften angefragt werden bzw. auf den Werten von diesen gefiltert wird.

In Kombination mit den Erkenntnissen, die im Abschnitt 3.1.4 aus den bereits existierenden Speichermodellen gewonnen wurden, wurde ein natives Speichermodell entwickelt, das die RDF-Daten so auf Datenbankseiten verteilt, dass Ressourcen zusammen mit ihren Eigenschaften und Werten mit wenigen Leseoperationen geladen werden können.

3.2.2 Physische Datenverwaltung

Das aus relationalen Datenbanksystemen bekannte und bewährte Konzept der Verwaltung der Daten auf Datenbankseiten – im Folgenden kurz als *Seite* bezeichnet – wird im beschriebenen Speichermodell beibehalten, jedoch wird die Zuordnung von Daten zu einer Seite sowie die Organisation der Daten auf einer Seite geändert. Der grundlegende Gedanke bei der Verwaltung von RDF-Daten ist, dass jede Ressource in der Rolle als Subjekt zusammen mit ihren Eigenschaften und Werten auf einer einzigen Datenbankseite verwaltet wird, jedoch ist kein fester Speicherbereich für sie auf der Seite reserviert.

Ersteres verfolgt zwei Ziele: Zum einen steht aufgrund der Eindeutigkeit einer IRI im RDF-Graphen die Datenbankseite fest, auf der eine Ressource verwaltet wird. Zum anderen werden der Bearbeitung einer Anfrage weniger Join-Operationen benötigt als bei tripelorientierten Speichermodellen (vgl. schemaunabhängige Speichermodelle auf Seite 18). Letztendlich bedeutet diese Anordnung, dass der Join über das Subjekt bereits vorberechnet ist.

Letzteres ermöglicht die Speicherung von mehrwertigen Attributen und vermeidet die Notwendigkeit, aufgrund von nicht belegten Eigenschaften bei einer Ressource Nullwerte speichern zu müssen (vgl. *schemaabhängige Speichermodelle* auf Seite 19).

Des Weiteren kann die Typinformation zu einer Ressource für ein initiales Gruppieren von Ressourcen ausgenutzt werden. Das Ausnutzen dieser Information ist sinnvoll, da Ressourcen im Allgemeinen dieselben oder ähnlich Eigenschaften besitzen und sehr wahrscheinlich gleichzeitig angefragt werden.

Im Folgenden wird die obige Abbildung formal beschrieben, wobei zunächst werden die Begriffe Datenbankseite[4] und Datenbank eingeführt. Dar-

[3] http://www.informatik.uni-trier.de/~ley/db/

[4] In der Literatur wird eine Datenbankseite auch als *Block* bezeichnet.

auf aufbauend wird anschließend die Abbildung zwischen einem RDF-Graphen und Datenbank definiert.

Definition 3.1 (Datenbankseite)**.** *Eine Datenbankseite, in dieser Arbeit mit* p *bezeichnet, ist ein persistenter Speicherbereich mit einer fest definierten Größe* $\|\mathsf{p}\|$*. Mit dem Symbol* $\mathcal{P}$ *wird die (theoretisch) unendliche Menge aller zur Verfügung stehenden Datenbankseiten bezeichnet.*

Auf Datenbankseiten werden später die Tripel von RDF-Graphen verwaltet. Jede Seite ist durch eine eindeutige ID identifizierbar . Als Konvention beschreibt die Notation $t \in \mathsf{p}$ ein Tripel $t \in G$, das auf der Seite p gespeichert ist. Weiterhin bezeichnet $\|\mathsf{p}\|$ die Seitengröße in Bytes und $|\mathsf{p}|$ die Anzahl der Tripel auf einer Seite. Es werden die Schreibweisen $\|t^s\|, \|t^p\|$ und $\|t^o\|$ eingeführt, mit denen der benötigte Speicherplatz in Bytes für das Subjekt, Prädikat bzw. Objekt eines Tripels t auf einer Datenseite ausgedrückt wird.

Definition 3.2 (Datenbank)**.** *Eine Datenbank* $\mathcal{D}$ *ist als eine Menge von Datenbankseiten definiert:* $\mathcal{D} = \{\mathsf{p} : \mathsf{p} \in \mathcal{P}\}$.

Aus der obigen Definition kann abgeleitet werden, dass ein Hinzunehmen einer neuen, leeren Seite zu einer Datenbank immer möglich ist. Darüber hinaus ist eine Datenbank genau dann leer ($\mathcal{D} = \emptyset$), wenn sie keine Seiten enthält. Als Konvention bezeichnet $|\mathcal{D}|$ die Anzahl der Datenbankseiten die Datenbank $\mathcal{D}$ enthält.

Die Abbildung einer Menge von Graphen auf eine Datenbank bzw. Datenbankseiten ist durch die Definition 3.3 gegeben. Bei dieser Definition findet die Normalisierung von IRIs und Literalen noch keine Berücksichtigung, sondern wird im Abschnitt 3.2.3 aufgegriffen.

Definition 3.3 (Speicherabbildung)**.** *Seien* $G_1, \ldots, G_n \in \mathcal{G}$ *beliebige RDF-Graphen und* $\mathcal{D}$ *eine Datenbank. Die Graphen werden nach den folgenden Kriterien in der Datenbank gespeichert:*

1. *Alle Tripel* t *einer Datenbankseite* p *gehören zu demselben Graphen* G_i*:*

$$\forall \mathsf{p} \in \mathcal{D} \forall t_i, t_k \in \mathsf{p} \exists G_i \in \{G_1, \ldots, G_n\} : t_i \in G_i \wedge t_k \in G_i$$

2. *Alle Subjekte* t^s *einer Datenbankseite* p *sind von einem Typ:*

$$\forall \mathsf{p} \in \mathcal{D} \forall t_i, t_k \in \mathsf{p} : \tau(t_i^s) = \tau(t_k^s)$$

3. *Alle Tripel* t *mit demselben Subjekt* t^s *liegen auf ein und derselben Datenbankseite* p*:*

$$\forall G_i \in \{G_1, \ldots, G_n\} \forall t_i, t_k \in G_i : t_i^s = t_k^s \rightarrow \exists \mathsf{p} \in \mathcal{D} : t_i \in \mathsf{p} \wedge t_k \in \mathsf{p}$$

4. *Zur Normalisierung wird eine eineindeutige Funktion* $\iota : \mathbb{I} \cup \mathbb{L} \rightarrow \mathbb{N}$ *verwendet, die IRIs und Literale auf eine natürliche Zahl abbildet.*

An dieser Stelle sei bereits darauf hingewiesen, dass das Kriterium 3 nicht für beliebige RDF-Graphen werden kann, da eine Datenbankseite eine begrenzte Kapazität besitzt. Wie später im Kapitel 3.2 illustriert wird, ist ein Überlauf einer Datenbankseite eher ein pathologischer Fall.

Abbildung 3.11 illustriert die Abbildung von den RDF-Aussagen des fortlaufenden Beispiels auf Datenbankseiten. Die Tripel werden nach der Normalisierung (s. Tabelle in der Abbildung) entsprechend des Typs ihres Subjekts auf verschiedene Seiten verteilt. Zum Beispiel enthält die Seite mit der ID 2 alle Tripel zu den Ressourcen `ex:proc1` und `ex:proc2`. Da die IDs der Ressourcen sowohl als Subjekt als auch als Objekt verwendet werden, werden implizit Verknüpfungen zwischen Datenbankseiten hergestellt. Im gezeigten Beispiel referenzieren die Ressource 3 auf Seite 2 (`ex:pers1`) und die Ressource 3 auf Seite 1 dieselbe Ressource des RDF-Graphen.

Normalisierung

ID	Wert
1	`ex:proc1`
2	`ex:proc2`
3	`ex:pers1`
4	`ex:pers2`
5	`foaf:person`
6	`"John Due"`
7	...

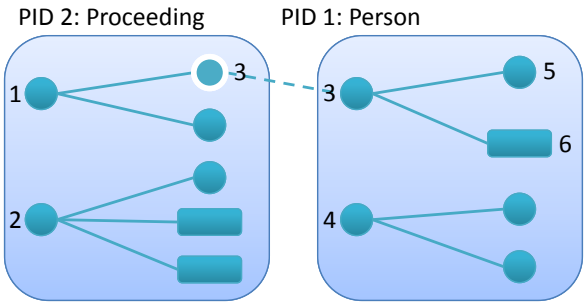


Abbildung 3.11: Abbildung eines RDF-Graphen auf Datenbankseiten (Eigenschaft nicht eingetragen)

Wie die schematische Darstellung einer Datenbankseite in Abbildung 3.12 (links) verdeutlicht, ist die interne Struktur in die Bereiche Kopf und Rumpf aufgeteilt. Der Kopf einer Seite enthält Metainformationen wie die ID der Seite, die Typen der gespeicherten Subjekte sowie die Anzahl der Subjekte. Im Rumpf einer Seite werden die Tripel verwaltet, wobei die Tripel in seine Komponenten aufgeteilt gespeichert werden. Die Komponenten eines Tripels werden mit deren IDs referenziert. Im Tripelindex wird der Zusammenhang zwischen den Einträgen im Bereich Subjekt und den zugehörigen Prädikaten und Objekten festgehalten. Dabei beinhaltet der erste Eintrag im Tripelindex die Adresse des ersten Prädikats bzw. Objekt des ersten Subjekts, der zweite Eintrag die Adresse für das erste Prädikat bzw. Objekt des zweiten Subjekts, usw. In Abbildung 3.12 (rechts) beispielsweise beginnen die Prädikat-/Objekt-IDs von Subjekt s_2 an Position 2 und enden an 4. Aufgrund der Struktur der Seite müssen Subjekte nicht wiederholt werden und alle Subjekt, Prädikate bzw. Objekte können effizient durchlaufen werden.

Für die Normalisierungsfunktion von Ressource-IRIs und Literalen wurde ein einfache Abbildung gewählt, die jedem neuen Wert die nächsthöhere, noch nicht vergebene natürliche Zahl als ID zuweist. Die Zuordnung von ID zu IRI bzw. Literal wird in einem Index (B+-Baum) gespeichert, der primär ein effizientes Nachschlagen von IDs unterstützt. Diese Richtung des Nachschlagens zu bevorzugen, ist sinnvoll, da am Ende der Anfragebearbeitung potentiell eine große Menge der intern zur Berechnung der Lösungsabbildungen verwendeten IDs wieder zu IRIs und Literalen aufgelöst werden müssen.

Für die Evaluation des Resource Centered Stores ist eine Speicherung der Normalisierungsabbildung in einem B+-Baum ausreichend. Eine Verwendung einer alternativen Indexstruktur oder Normalisierungsfunktion wäre jedoch

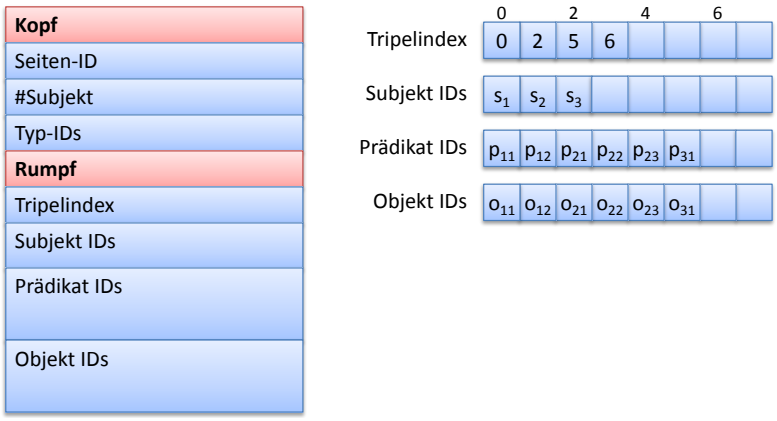


Abbildung 3.12: Interne Struktur einer Datenbankseite: schematische Darstellung (links) und Detailansicht des Rumpfes (rechts)

sinnvoll, wenn insbesondere die SPARQL-Funktionen und Prädikate unterstützt werden sollen, die über den Zeichenketten von IRIs und Literalen operieren. Beispielsweise lassen sich mit dem verwendeten B+-Baum Filterausdrücke mit einer Bereichsanfrage wie `?x > "Müller"` oder einer Zeichenkettenfunktion wie `contains("ller")` nicht effizient auswerten.

3.2.3 Zugriffsstrukturen

Neben der Zuordnung von Tripeln zu Datenbankseiten werden Zugriffsstrukturen benötigt, die ein schnelles Wiederauffinden von Tripel in der Datenbank erlauben. Zu diesem Zweck werden Indexe auf den jeweiligen Komponenten eines Tripels aufgebaut, die zu einem gegebenen Subjekt, Prädikat oder Objekt die Adresse der Tripel auf den Datenbankseiten zurückgeben, in denen diese vorkommen. Diese Indexe werden in den nachfolgenden Abschnitten respektive als *Subjekt-*, *Prädikat- und Objektindex* bezeichnet. Ein Index auf den Typen der Datenbankseiten wird nicht separat gehalten, da dieser durch das Kombinieren von Prädikat- (auf `rdf:type`) und Objektindex (der jeweiligen Typressource) simuliert werden kann.

Für die Subjekte der Tripel kann aufgrund der Eindeutigkeit von IRIs im RDF-Graphen ein Index von einer Ressource auf die Adresse des Subjekts auf der zugehörigen Datenbankseite genutzt werden (s. u.). Letztendlich wird mit dieser Adresse die Menge aller Tripel zu einem gegebenen Subjekt adressiert. In den nachfolgenden Abschnitten wird die Adresse eines Tripels/Subjekts auf einer Datenbankseite auch als *Slot* bezeichnet. Bei dieser Form der Adressierung wird davon ausgegangen, dass ein Subjekt zusammen mit allen zugehörigen Tripeln auf eine Datenbankseite passen. Diese Annahme ist angesichts der durchschnittlichen Anzahl von 25 Tripeln pro Subjekt im Vergleich zu den maximal ca. 8,000 Tripel pro Datenbankseite der Größe 32 kB durchaus realistisch (vgl. Abschnitt 3.3). Im seltenen Fall des Überschreitens der Kapazität einer Datenbankseite wird mit Überlauf-Datenbankseiten gearbeitet und ein erhöhter Aufwand bei der Anfragebearbeitung in Kauf genommen. Sofern nicht gesondert vermerkt, wird daher im restlichen Teil der Arbeit davon aus-

gegangen, dass ein Subjekt zusammen mit all seinen Tripeln auf einer einzelnen Datenbankseite gespeichert werden können.

Um die Subjekte mit bestimmten Prädikaten oder Objekten zu ermitteln, werden Bitset-Indexe verwendet. Gegenüber von B-Bäumen haben diese den Vorteil, dass sich Bitsets bei mehrdimensionalen Punktanfragen, wie sie in SPARQL-Anfrage häufig zu erwarten sind, einfacher miteinander kombinieren lassen (Verundung) als B-Bäume. Zu jeder Ressource in der Rolle als Prädikat oder Objekt bzw. jedem Literal wird ein Bitset erzeugt. Ein effizienter Zugriff auf das zu einem Prädikat oder Objekt gehörende Bitsets wird durch zwei B+-Bäume gewährleistet.

Eine Position i in einem Bitset wird dabei als Kombination aus Datenbankseiten-ID und Adresse eines Subjekts (Slot) auf dieser Seite interpretiert. Bei der Kodierung werden die ersten n Bits als Slot und die restlichen Bits als Seiten-ID interpretiert (vgl. Abbildung 3.13).

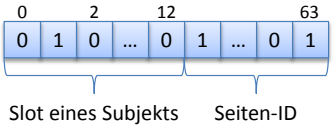


Abbildung 3.13: Bitset-Position kodiert den Slot eines Subjekts auf einer Seite und deren ID

Wenn beispielsweise eine Position im Bitset durch 64 Bit repräsentiert wird, dann können $2^{12} = 4096$ Adressen (entspricht einer Seitengröße von 32 kB) und $2^{52} \approx 4 \cdot 10^{15}$ Seiten-IDs adressiert werden. Aufgrund der eindeutigen Zuordnung eines Subjekts zu einer Datenbankseite kann ein bestimmtes Tripel mit gegebenen Subjekt nur auf einer Seite auftreten; somit ist obige Kodierung ebenfalls eindeutig. Ein gesetztes Bit an der Position i bedeutet hierbei, dass das sich aus Kodierungsvorschrift ergebene Tripel das gegebene Prädikat/Objekt besitzt. Die Bitsets werden mit einem gängigen Verfahren wie [LKA10] komprimiert gespeichert. Kompressionsverfahren erreichen die größte Kompression, wenn besonders lange Läufe von Nullen auftreten und die Daten sortiert vorliegen. In Abschnitt 3.3 wird eine Abschätzung der Größen der Bitset-Indexe vorgenommen.

Die zuvor beschriebenen Indexstrukturen sind besonders gut geeignet, konkrete Ressourcen und Literale auf den Datenbankseiten zu lokalisieren. Das heißt, sie ermöglichen insbesondere das Auswerten von Basis-Graphmustern. Wenn darüber hinaus auch die Auswertung von Filterausdrücken (z. B. Bereichsanfragen und Textsuche) unterstützt werden sollen, dann sind zusätzliche Berechnungsschritte erforderlich. Diese werden in Abschnitt 5.4.2 näher betrachtet.

3.2.4 Operationen

Auf Grundlage der obigen Abbildung von Tripeln auf Datenbankseiten und die Indexstrukturen wird im Folgenden beschrieben, wie die Operationen Anfragen, Hinzufügen und Löschen von Daten auf dem Speichermodell ausgeführt werden. Das Aktualisieren von Daten kann als Löschen des zu ändernden Tripels und das Hinzufügen eines Tripels mit den neuen Werten betrachtet werden – in einer Implementierung würde dafür ein optimierter Algorithmus umgesetzt werden.

Bezüglich der Auswertung von Anfragen werden im Folgenden nur die grundlegenden Algorithmen betrachtet. Alternative Evaluationsstrategien für

SPARQL-Anfragen werden in Abschnitt 5.4 beschrieben und diskutiert (z. B. eine kombinierte Auswertung von Basis-Graphmustern und Filterausdrücken).

Bearbeitung von Anfragen Ein Basis-Graphmuster kann als die Verknüpfung von sternförmigen Teilmuster (vgl. Definition 2.12 auf Seite 14) betrachtet werden. Im Folgenden wird zunächst ein Algorithmus vorgestellt, mit dem die Variablenbindungen eines sternförmigen Basis-Graphmusters berechnet werden können. Anschließend wird die Berechnung der Variablenbindungen für ein allgemeines Basis-Graphmuster vorgestellt.

Mit Algorithmus 3.1 wird zunächst nur der Fall eines Basis-Graphmusters in Sternform betrachtet. Wie in Abbildung 2.4 auf Seite 14 illustriert, können grundsätzlich zwischen zwei Anfragetypen unterschieden werden: (1) Anfragen mit gegebenen Subjekt und (2) Anfragen mit einer Variablen als Subjekt.

Bei Anfragen des ersten Typs (Zeilen 4–6) wird über den Subjektindex die Speicheradresse (Datenbankseite und Slot) der Tripel mit dem entsprechenden Subjekt ermittelt. Die adressierte Datenbankseite wird vom Sekundärspeicher geladen und anhand der Slot-Information die zum Subjekt gehörenden Tripel bestimmt. Durch einen abschließenden Abgleich mit dem Basis-Graphmuster werden gegebenenfalls Variablenbindungen generiert.

Im zweiten Fall (Zeilen 8–22) werden die zu dem Prädikat und/oder Objekt gehörenden Bitsets aus dem Prädikat- und Objektindex geladen und gegebenenfalls mit einem bitweisen Und verknüpft. Aus dem (resultierenden) Bitset werden die zu ladenden Seiten ermittelt (Zeile 19), die dann nacheinander geladen und auf das Basis-Graphmuster hin durchsucht werden. Für jeden passenden Teilgraphen werden Variablenbindungen erzeugt.

Im Algorithmus 3.2 wird ein naïver Ansatz zum Bestimmen der zu ladenden Datenbankseiten gezeigt. Mit Hilfe von statistischen Informationen über die Verteilung von Prädikate und Objekt auf den Seiten könnten nur die besonders selektiven Bitsets berücksichtigt werden. „Besonders selektiv“ heißt hierbei, dass nur wenige Datenbankseiten durch das Bitset selektiert werden.

Die Beantwortung von sternförmigen Anfragen bildet die Basis für das Berechnen der Lösungen von allgemeinen Basis-Graphmustern. Ein trivialer Ansatz (Algorithmus 3.2) dafür zerlegt die Anfrage in ihre sternförmigen Teilanfragen – manche können nur ein Tripel groß sein – und verknüpft deren Lösungen basierend auf den Join-Variablen der Anfrage. Aus der Lösung einer Teilanfrage können unter Umständen auch Rückschlüsse auf das Berechnen von anderen gezogen werden. Eine Anfrage wie in Abbildung 3.14 kann in zwei Teilanfragen bestehend aus jeweils einem Tripelmuster zerlegt werden. Anstatt die Lösungen für jedes Tripelmuster einzeln zu berechnen, könnte zunächst das Tripelmuster (1) berechnet werden und die Variablenbindungen für `?person` zusammen mit dem Subjektindex für das Ermitteln der Namen genutzt werden. Wenn das zweite Tripelmuster besonders selektiv wäre, könnten umgekehrt die Werte für `?person` in Kombination mit dem Objektindex verwendet werden. Die Entscheidung über die richtige Evaluierungsstrategie wird vom Anfrageoptimierer auf Basis von statistischen Informationen getroffen (vgl. Abschnitt 5.5).

Grundsätzlich können Filterausdrücke einer Anfrage nach dem Berechnen der Lösungen für ein Graphmuster ausgewertet werden. Jedoch können diese auch genutzt werden, um die Anzahl der zu ladenden Datenbankseiten zu re-

Algorithm 3.1 matchStarBGP(BGP bgp) : Lösungsabbildungen

```
1:  bindings = ∅
2:  tp = bgp.first                                   // das erste Tripelmuster
3:  if (isResource(tp.subject)) then
4:     address = subjectIdx.get(tp.subject)
5:     page = load(address.pageId)
6:     bindings = match(page, address.slot, tp)
7:  else
8:     bitset = ∼ 0                                  // Bitset mit allen Bits gesetzt
9:     for TriplePattern tp : bgp do
10:       if (isResource(tp.predicate)) then
11:          bitset &= predicateIdx.get(tp.predicate)
12:       end if
13:       if (isResource(tp.object) || isLiteral(tp.object)) then
14:          bitset &= objectIdx.get(tp.object)
15:       end if
16:    end for
17:
18:    addresses = subjectIdx.get(bitset)   // Seiten-IDs der Subjekte ermitteln
19:    for address : addresses do
20:       page = load(address.pid)
21:       bindings += match(page, address.slot, tp)
22:    end for
23: end if
24: return bindings
```

```
1 {   ?proceeding dcterms:editor ?person .
2     ?person foaf:name ?name . }
```

Abbildung 3.14: Beispiel für die einfachste Anfrage bestehend aus zwei sternförmigen Teilanfragen

Algorithm 3.2 matchBGP(BGP bgp) : Lösungsabbildungen

```
1:  bindings = ∅
2:  stars = decompose(bgp)                          // in Sternanfragen zerlegen
3:  for BasicGraphPattern p : stars do
4:     b = matchStarBGP(p)
5:     if (bindings = ∅) then
6:        bindings = b
7:     else
8:        bindings = bindings ⋈ b
9:     end if
10: end for
11: return bindings
```

duzieren. Treten beispielsweise in einer Anfrage Filterausdrücke über Literale auf, so werden aus den Literalindexe die Adressen der passenden Tripel ermittelt, ein Bitset daraus erzeugt und mit den für die Anfrage relevanten Bitsets aus dem Prädikat- und Objektindex verknüpft. Spezielle Ausführungsstrategien dafür werden im Abschnitt 5.4.2 beschrieben.

Hinzufügen eines Tripels Bevor ein Tripel in der Datenbank gespeichert wird, wird eine Normalisierung der IRIs und Literale durchgeführt. Anschließend wird zunächst der Typ des Subjekts bestimmt: Wenn das Prädikat des Tripels `rdf:type` ist, dann ist der Typ gleich dem Objekt des einzufügenden Tripels; ansonsten ist er `rdf:Resource` (Zeilen 1–4).

Beim Hinzufügen eines Tripels t zu einem RDF-Graphen (vgl. Algorithmus 3.3) hängt die Vorgehensweise davon ab, ob bereits ein Tripel mit dem Subjekt t^s in der Datenbank existiert. In Falle der Existenz (Zeilen 7–13) wird die zum Subjekt gehörende Datenbankseite geladen und das Tripel, falls es dieses dort noch nicht gibt, auf dieser abgelegt. Ein Sonderfall tritt auf, wenn das neue Tripel den Typ der Ressource definiert, d.h. das Prädikat des Tripels ist `rdf:type`. Damit ist der Typ des Subjekts die Vereinigung von dem aus dem Tripel gegebenen Typ und den Typen der Datenbankseite ($\tau(t^s) = t^o \cup \tau(\mathrm{p})$). Aufgrund der Abbildungsvorschrift in Definition 3.3 muss nun überprüft werden, ob das Subjekt auf dieser Seite verbleiben kann, eine andere Seite mit passendem Typ und genügend Kapazität gefunden[5] oder eine neue Seite für diesen Typ angelegt werden muss.

Wenn das Subjekt des hinzuzufügenden Tripels bislang noch nicht in der Datenbank existiert, wird anhand des Typs der Ressource eine Datenbankseite dieses Typs mit genügend Kapazität gesucht (Zeilen 15–16). Das Tripel wird dann auf diese Datenbankseite abgelegt. Auch hier wird nach einer Datenbankseite passenden Typs gesucht und ggf. eine neue angelegt.

Unabhängig davon, ob das Subjekt bereits in der Datenbank existiert, kann ein Tripel nur zu einer Seite hinzugefügt werden, wenn dort Speicherplatz zur Verfügung steht. Gegebenenfalls muss eine neue Datenbankseite alloziert werden, um durch Verschieben von Tripeln genügend Platz auf der Seite zu schaffen. Um einen minimalen Füllgrad einer Datenbankseite sicherzustellen, können auch mehrere Subjekte verschoben werden (ausbalancieren, Zeilen 12 und 17).

Des Weiteren müssen im Anschluss an das Hinzufügen eines Tripels alle Indexe aktualisiert werden. Falls mit dem Tripel ein bisher nicht vorhandenes Prädikat oder Objekt hinzugefügt worden ist, müssen beispielsweise im Prädikat- und Objektindex neue Bitsets erzeugt werden. Ansonsten müssen die Bits für die vom Einfügen betroffenen Seiten aktualisiert werden, da Tripel auf den Seiten verschoben worden sein könnten.

Löschen eines Tripels Beim Löschen eines Tripels (vgl. Algorithmus 3.4) wird zunächst mittels des Subjektindexes die betroffene Datenbankseite bestimmt und geladen; das Tripel wird anschließend von der Datenbankseite gelöscht (Zeilen 3 und 4). Um eine hinreichenden Füllgrad der Seiten beizubehalten, kann nach dem Löschen ein Ausbalancieren von Seiten ausgeführt werden (Zeile 8). Gegebenenfalls werden bei diesem Schritt leere Seiten aus

[5] Über den Prädikatindex lassen sich die Kandidatenseiten effizient bestimmen.

Algorithm 3.3 insertTriple(NormalizedTriple t) : void

```
1:  type = rdf:resource
2:  if (t.predicate = rdf:type) then
3:     type = t.object
4:  end if
5:  pageId = subjectIdx.get(t.subject)
6:  if (pageId) then
7:     page = load(pageId)
8:     addTriple(page, t)
9:     if (type ∉ τ(page)) then
10:       restoreTypeContraint(page, t)        // Gleichzeitig wird ausbalanciert
11:    else
12:       balancePage(page)
13:    end if
14: else
15:    page = findOrAllocatePage(type)
16:    addTriple(page, t)
17:    balancePage(page)
18: end if
```

der Datenbank entfernt. Wie auch beim Hinzufügen muss beim Löschen eines Tripels mit dem Prädikat `rdf:type` überprüft werden, ob weitere Aktionen wie das Verlegen des Subjekts auf eine andere, zum Typ passende Datenbankseite erforderlich ist (Zeile 6).

Im Anschluss an das Löschen eines Tripels müssen alle Indexe aktualisiert werden. Beispielsweise kann im Prädikat- und Objektindex ein Bitset aus den Indexen entfernt werden, wenn mit dem Tripel das letzte Vorkommen eines Prädikats oder Objekts gelöscht worden ist. Alternativ müssen die Bits für die vom Löschen betroffenen Seiten aktualisiert werden, da auf den Seiten einige Tripel verschoben worden sein könnten.

Algorithm 3.4 deleteTriple(NormalizedTriple t) : void

```
1:  pageId = subjectIdx.get(t.subject)
2:  if (pageId) then
3:     page = load(pageId)
4:     deleteTriple(page, t)
5:     if (t.predicate = rdf:type) then
6:        restoreTypeContraint(page, t)        // Gleichzeitig wird ausbalanciert
7:     else
8:        balancePage(page)                    // Löscht leere Seiten
9:     end if
10: end if
```

3.3 Analyse des RCS-Speichermodells

Auf Basis des zuvor beschriebenen Speichermodells werden in diesem Abschnitt Formeln für das Abschätzen des Speicherplatzbedarfs für das Spei-

chern der Daten als auch für die Indexe herausgearbeitet und anhand von statistischen Daten über DBpedia veranschaulicht. Abschließend wird es mit der Komplexitat anderer existierender Speichermodellen verglichen.

3.3.1 Daten

Die minimale Anzahl der benötigten Datenbankseiten für das Speichern eines RDF-Graphen hängt weniger von der Tripelanzahl sondern von der Anzahl der vorkommenden Typen der Ressourcen ab, da für jeden Typ mindestens eine eigene Datenbankseite alloziert werden muss (vgl. Definition 3.3).

Für das Berechnen der maximalen Anzahl an Datenbankseiten, die für das Speichern eines RDF-Graphen benötigt wird, wird zunächst eine Abschätzung für die Anzahl der im Rumpf einer Seite speicherbaren Tripel benötigt. In den unten stehenden Berechnungen wird der Anteil des Kopfes einer Seite vernachlässigt, da dessen Größe im Wesentlichen von der Anzahl der Typ-IDs abhängt, die jedoch im Allgemeinen drei nicht übersteigt.

Die Anzahl der speicherbaren Tripel ergibt sich somit aus dem Verhältnis der Größe einer Datenbankseite $\|\mathsf{p}\|$ und der Größe des benötigten Speicherplatzes für die Tripel $m\|t^s\|+n(\|t^p\|+\|t^o\|)$, wobei m die Anzahl der Subjekte und n die Anzahl der Tripel auf der Seite ist.[6] Da Subjekte nicht wiederholt gespeichert werden, gibt es für die Anzahl der Tripel einen minimalen (Formel 3.1) und maximalen Wert (Formel 3.2) – alle Tripel haben ein unterschiedliches bzw. dasselbe Subjekt. Daraus ergeben sich unten stehende Formeln, wenn der Speicherplatz der Datenbankseite voll ausgeschöpft wird.

$$
\begin{aligned}
& & \|\mathsf{p}\| &= m\|t^s\| + n(\|t^p\| + \|t^o\|) & \\
& \overset{m=n}{\Longrightarrow} & n_{min} &= \frac{\|\mathsf{p}\|}{\|t^s\| + \|t^p\| + \|t^o\|} & (3.1)\\
& \overset{m=1}{\Longrightarrow} & n_{max} &= \frac{\|\mathsf{p}\| - \|t^s\|}{\|t^p\| + \|t^o\|} & \\
& \overset{\|\mathsf{p}\| \gg \|t^s\|}{\Longrightarrow} & n_{max} &\approx \frac{\|\mathsf{p}\|}{\|t^p\| + \|t^o\|} & (3.2)
\end{aligned}
$$

In Tabelle 3.2 wurden zur Veranschaulichung die ungefähren minimalen und maximalen Werte für die Tripelanzahl bezüglich unterschiedlicher Seitengrößen berechnet, wobei die Größe einer ID vier bzw. acht Byte und die Größe eines Eintrags im Tripelindex zwei Byte beträgt. Aus der Tabelle ist abzulesen, dass mit einer Verdopplung der Seitengröße sich die Werte für n_{min} und n_{max} ebenfalls annähernd verdoppeln.[7]

Obige Abschätzung gegeben lassen sich für einen gegebenen RDF-Graphen G eine minimale und maximale Anzahl von Seiten für die Datenbank bestimmen (Gleichungen 3.3 und 3.4). Des Weiteren ergibt sich selbst für sehr kleine Graphen eine untere Schranke (Gleichung 3.5), die sich aus der Anzahl unterschiedlicher Typen von Ressourcen ergibt; für jeden Typ einer Ressource ist eine Datenbankseite erforderlich.

[6] Die Größe des Subjekts $\|t^s\|$ beinhaltet auch die Größe des Eintrags im Tripelindex

[7] Annähernd, weil der für Tripel nutzbare Bereich aufgrund des günstigeren Verhältnisses zwischen dem Speicherplatz für Kopf und Rumpf steigt.

$\|\|ID\|\|$	$\|\|\mathsf{p}\|\|$	8 kB	16 kB	32 kB	64 kB
4 Byte	$\sim n_{min}$	500	1.100	2.300	4.600
	$\sim n_{max}$	1.000	2.000	4.000	8.000
8 Byte	$\sim n_{min}$	300	600	1.200	2.500
	$\sim n_{max}$	500	1.000	2.000	4.000

Tabelle 3.2: Anzahl der Tripel pro Datenbankseite

$$|\mathcal{D}|_{min} = \sum_{k \in \tau(G)} \frac{|\{t \in G : \tau(t^s) = k\}|}{n_{max}} \tag{3.3}$$

$$|\mathcal{D}|_{max} = \sum_{k \in \tau(G)} \frac{|\{t \in G : \tau(t^s) = k\}|}{n_{min}} \tag{3.4}$$

$$|\mathcal{D}|_{lb} = |\tau(G)| \tag{3.5}$$

Zusammengefasst kann die Hypothese aufgestellt werden, dass die Anzahl der belegten Datenseiten und damit die Größe der Datenbank proportional zu der Anzahl der Tripelmuster wächst.

3.3.2 Zugriffsstrukturen

Im Folgenden wird der Speicherplatzbedarf der zwei im Speichermodell verwendeten Arten von Indexen (B+-Baum und Bitset) abgeschätzt. Die B+-Bäume werden zum einen zur Verwaltung der Bitsets und zum anderen für die Literalindexe verwendet. Basierend auf [BM70] ergibt sich die folgende Abschätzung für die Anzahl der Seiten für einen B+-Baum, wobei jeder Knoten n_i des Baums auf einer Seite abgelegt wird.

$$e = \left\lfloor \frac{\|\mathsf{p}\|}{\|key\| + \|pointer\|} \right\rfloor \quad \text{Anzahl der Indexeinträge pro Seite}$$

$$b = \left\lfloor \frac{\|\mathsf{p}\| + \|key\|}{\|pointer\|} \right\rfloor \quad \text{Verzweigungsfaktor}$$

$$n_0 = \left\lceil \frac{N}{e} \right\rceil \quad \text{Anzahl der Blätter im Baum}$$

$$n_i = \left\lceil \frac{n_{n-1}}{b} \right\rceil \quad \text{Anzahl der Knoten auf Ebene } i$$

Demzufolge ist die Komplexität der Speicherbedarfs $O(N)$, wobei N die Anzahl der indexierten Werte ist. Die Werte $\|key\|$ und $\|pointer\|$ bezeichnen den Platzbedarf für den Schlüssel beziehungsweise für den Verweis auf die Daten; diese hängen von der jeweiligen Implementierung ab.

Betrachtet man zum Beispiel die Anzahl der Entitäten in DBpedia [BLK+09] von $2,8 \cdot 10^6$, so werden unter der Annahme einer Seitengröße von 16 kB und einer Größe des Schlüssels und Pointers von jeweils 8 Byte ca. 2.740 Seiten und damit rund 42 MB benötigt. Für einen B+-Baum über alle Tripel $70,2 \cdot 10^6$ von

DBpedia werden unter denselben Annahmen 68.590 Seiten und rund 1 GB Speicher benötigt. Bei einer Verdoppelung der Größe von Schlüssel und Pointer verdoppelt sich der Speicherplatzbedarf für den B+-Baum. In beiden Fällen hat der Baum eine Höhe von drei und die ersten beiden Ebenen können auf jeden Fall ständig im Hauptspeicher gehalten werden.

Für die Abschätzung der Speicherkosten eines Bitsets wird der nach WAH komprimierte Bitset [WOS06] als Grundlage genommen. In diesem Index werden die Bits wortweise komprimiert, wobei ein Wort aus 32 oder 64 Bit besteht. Für den schlechtesten Fall (*engl. worst case*), bei dem alle Bits gesetzt sind, wird in [WOS06] die Größe eines Bitsets mit $2N$ Worten abgeschätzt, im Kontext des RCS-Speichermodells N die Anzahl der Tripel ist. Jedoch ist bei RDF-Daten davon auszugehen, dass für fast alle Eigenschaften und Ressourcen dieser Fall nicht eintritt und die Größe eines Bitset wesentlich geringer ist. Wenn man die Größe eines Bitsets in der Anzahl benötigter Seiten ausdrückt, so ergibt sich bei einer Wortlänge von 4 Byte (32 Bit):

$$|\mathsf{p}_{Bitset}| = \frac{4 \cdot 2N}{||\mathsf{p}||}$$

Die Verwendung von weiteren Kompressionsstrategien ist denkbar, mit denen Speicherplatzbedarf gegen Performanz aufgewogen wird, z. B. k-von-N-Kodierung (*engl. K-of-N encoding*). Diese Strategien werden aber im Rahmen dieser Arbeit nicht näher betrachtet. Um diese Größe ins Verhältnis zu denen von B+-Bäumen zu stellen, führen Wu et al. als Beispiel eine nicht näher beschriebene kommerzielle Implementierung eines B+-Baums mit einem Speicherbedarf von $3N \sim 4N$ Wörter an.

Nicht zuletzt hängt die Anzahl der zu verwaltenden Bitsets von den Anzahl der distinkten Eigenschaften und Objekten ab, da jeweils ein Bitset erforderlich ist. Demzufolge wird $O(|P|N + |O|N)$ Speicherplatz benötigt, wobei $|P|$ und $|O|$ die Anzahl der Eigenschaften bzw. der Objektressourcen bezeichnen. Lemire et al. stellen in [LKA10] fest, dass die Verwendung von Bitsets selbst für 100 Mio. Bitsets skaliert; somit ist der vorgeschlagene Ansatz auch für eine höhere Anzahl an Eigenschaften und Objekten geeignet.

Auch hierfür soll als Beispiel DBpedia herangezogen werden. Da für die Objekte keine Anzahl in einer Veröffentlichung gefunden werden konnte, wurde für den englischen Infobox-Teil von DBpedia in der Version 3.4 Statistiken berechnet. Bei einer Gesamtzahl von $44 \cdot 10^6$ Tripeln gibt es ca. 52.500 verschiedene Eigenschaften und $9,8 \cdot 10^6$ verschiedene Objekte ($5,9 \cdot 10^6$ Literale und $3,8 \cdot 10^6$ Ressourcen).

Da der Speicherbedarf für ein Bitset von der Verteilung der gesetzten Bits abhängig ist, wurde für alle Eigenschaften die jeweilige Anzahl der Tripel ermittelt. Nur für einen sehr geringen Anteil der Eigenschaften gab es mehr als 1.000 Tripel (vgl. Tabelle 3.3). Damit wird für den untersuchten Datensatz ein gutes Verhältnis zwischen Bitset-Länge und Anzahl gesetzter Einsen erreicht. Nimmt man den schlechtesten Fall für die Bitsets an – alles Einsen müssten mit 2 Byte repräsentiert werden –, so würde sich beispielsweise ein Index mit 1000 k Einsen bei einer Seitengröße von 32 kB eine Größe von 8 MB haben und sich über 250 Seiten erstrecken. Jedoch ist dies nicht der Regelfall, sondern fast alle Bitsets (ca. 98,2 %) könnten mit dem Zugriff auf eine Datenbankseite geladen werden.

Kardinalität	$> 1000k$	$500k$	$> 100k$	$> 50k$	$> 10k$	> 5000
abs. Anzahl	3	3	59	65	449	379
Prozentsatz	0,006 %	0,006 %	0,112 %	0,129 %	0,854 %	0,721 %
Kardinalität	> 1000	> 500	> 100	> 50	> 10	≤ 10
abs. Anzahl	1688	1212	6560	4361	9098	32546
Prozentsatz	3,209 %	2,304 %	8,292 %	5,191 %	17,298 %	61,879 %

Tabelle 3.3: Anzahl der Tripel für eine Eigenschaft

Da die Literale über B+-Bäume bzw. einen Volltextindex abgedeckt werden, werden für die Verwaltung der Objektressourcen des Infobox-Datensatzes von DBpedia insgesamt $3,8 \cdot 10^6$ Bitsets benötigt. Im Vergleich zu den Eigenschaften ist die Anzahl der Tripel mit derselben Objekt-Ressource wesentlich geringer (vgl. Tabelle 3.3). Der größte Index hat hier eine Größe von 0,8 MB und erstreckt sich bei einer Seitengröße von 32 kB über 25 Seiten. Wie auch bei den Analyse der Eigenschaften bleibt festzustellen, dass fast alle Bitsets (ca. 99,9 %) mit einem Seitenzugriff geladen werden können.

Kardinalität		$500k$	$> 100k$	$> 50k$	$> 10k$	> 5000
abs. Anzahl		0	10	12	105	125
Prozentsatz		0,000 %	$3 \cdot 10^{-4}$ %	$3 \cdot 10^{-4}$ %	0,003 %	0,003 %
Kardinalität	> 1000	> 500	> 100	> 50	> 10	≤ 10
abs. Anzahl	991	1228	10057	12010	93049	$3,8 \cdot 10^6$
Prozentsatz	0,026 %	0,032 %	0,261 %	0,311 %	2,412 %	96,953 %

Tabelle 3.4: Anzahl der Tripel für eine Objektressource

3.3.3 Operationen

Neben dem benötigtem Speicherplatz sind insbesondere die Aufwände für Operationen auf dem Speichermodell von Interesse und werden in diesem Abschnitt abgeschätzt. Dabei wird die Anzahl der Zugriffe auf den Sekundärspeicher (IO-Zugriffe) als Kriterium herangezogen.

Für die B+-Bäume [BM70] des Speichermodells kann zunächst eine Komplexität von $O(\log_b M)$ festgestellt werden, wobei b der Verzweigungsfaktor und M die Anzahl der indexierten Datensätze darstellt – im Kontext des Speichermodells sind dies Subjekte, Prädikate und Objekte.

Anfragen Für die Analyse der Komplexität von Anfragen an das Speichermodell werden die Algorithmen 3.1 und 3.2 herangezogen. Weiterhin werden nur Basis-Graphmuster betrachtet und keine weiteren Optimierungen wie die Verwendung von Statistiken über Eigenschaften und Ressourcen einbezogen.

Die Komplexität des Algorithmus 3.1, der das Finden von Lösungen zu sternförmigen Basis-Graphmustern beschreibt, hängt in erster Linie davon ab, ob das Subjekt gegeben ist. Falls es gegeben ist, so ist die Komplexität unabhängig von der Größe des Graphmusters immer $O(\log_b M)$, da nur der Subjektindex durchsucht und dann die entsprechende Datenseite geladen werden

muss. Aufgrund der Eigenschaften des Datenmodells enthält diese Seite alle zum Beantworten der Anfrage erforderlichen Tripel.

Ist das Subjekt eine Variable, sind für die Komplexitätsanalyse zwei Schritte zum Beantworten einer Anfrage Q wesentlich: (1) Identifizieren der zu ladenden Datenseiten und (2) Laden der Datenseiten. Die zu ladenden Seiten können über das Verunden der Bitsets von den konstanten Prädikate und Objekte der Anfrage identifiziert werden. Dazu müssen für jede Konstante die Bitsets über jeweils einen Zugriff auf den Prädikat- bzw. den Objektindex (B+-Bäume) lokalisiert werden. Die Komplexität für den schlechtesten Fall ist daher:

$$O(\underbrace{(|t_Q^p| + |t_Q^o|)\log_b M}_{\text{Zugriffe B+-Baum}} + \underbrace{(|t_Q^p| + |t_Q^o|)2N}_{\text{Zugriffe Bitsets}})$$

wobei $|t_Q^p|$ die Anzahl der konstanten Prädikate und $|t_Q^o|$ die Anzahl der konstanten Objekte bezeichnet. Die $2N$ entstammen der Komplexität für die Größe der Bitsets. Wie bereits im vorherigen Abschnitt anhand des DBpedia-Datensatzes aufgezeigt worden ist, ist dieser Wert fast immer sehr klein und im schlechtesten Fall würde für jeden Bitset jeweils eine Seite geladen werden müssen. Betrachtet man den Fall einer Variable als Subjekt, dann ist das Eintreten des schlechtesten Falls unwahrscheinlich, da Prädikate in einem Graphmuster wiederholt auftreten und die jeweiligen Bitsets bereits im Hauptspeicher vorliegen und kein Zugriff auf den Sekundärspeicher erforderlich ist. Für solche RDF-Daten wäre die Komplexität nur von der Anzahl der Konstanten in der Anfrage und nicht von der Anzahl der gespeicherten Tripel abhängig, d.h. $O(|t_Q^p| + |t_Q^o|)$.

Die Komplexität des zweiten Schritts (2) ist wegen der Konstruktion des Speichermodells unmittelbar von der Anzahl der Lösungen für das Graphmuster abhängig, wodurch sich eine Komplexität von $O(|S|)$ gibt, wobei $|S|$ der Anzahl an Lösungen entspricht. Damit ergibt sich für den schlechtesten Fall eine Komplexität für beide Schritte von:

$$O\Big((|t_Q^p| + |t_Q^o|)\log_b M + (|t_Q^p| + |t_Q^o|)2N + |S|\Big)$$

Der Algorithmus 3.2 bestimmt Lösungen für beliebige Basis-Graphmuster, wobei zunächst eine Anfrage Q in sternförmige Teilanfragen Q_i zerlegt wird, zu denen dann mittels der vorherigen Algorithmus Lösungen ermittelt werden. Diese werden anschließend miteinander verknüpft. Daraus ergibt sich eine Komplexität für den Zugriff auf Datenbankseiten von $O(nK_{\text{Alg 3.1}} + K_{\text{Join}})$, wobei n die Anzahl der sternenförmigen Teilanfragen, $K_{\text{Alg 3.1}}$ die Komplexität des Algorithmus' 3.1 und K_{Join} die Kosten für die Join-Operationen über den Teillösungen bezeichnen. Der für das Berechnen des Joins verwendete Algorithmus und damit die Kosten der Join-Operationen hängen von der Anzahl der Lösungen der Teilergebnisse ab – dem Anfrageprozessor werden somit weitere Optionen zum Optimieren der Anfrageausführung gegeben. Falls nicht alle Datenseiten im Hauptspeicher gehalten werden können, entstehen durch den Join zusätzliche Lese- und ggf. auch Schreiboperationen; die Kosten der Join-Algorithmen ist aus der relationalen Datenbanktheorie her bekannt. [ITL10] Durch die Zerlegung in sternförmige Anfragen hängt die Anzahl der benötigten Join-Operationen nicht mehr von der Anzahl der Tripelmuster sondern von der Anzahl der sternförmigen Teilanfragen ab.

Einfügen und Löschen Betrachtet man die beiden Operationen Einfügen und Löschen eines Tripels, so sind für die Komplexität die wesentlichsten Schritte das (1) Finden und Laden der Seite, auf der entweder das Tripel einzufügen bzw. zu löschen ist, (2) Aktualisieren des Subjektindexes, (3) Aktualisieren des Prädikatindexes, (4) Aktualisieren des Objektindex und (5) Schreiben der veränderten Datenbankseiten.

Aufgrund des B+-Baums auf dem Subjekt wird die Komplexität für (1) von dem Zugriff auf diesen dominiert und ist daher im schlechtesten Fall $O(\log_b M)$. Hinzu kommt das Laden und Schreiben der eigentlichen Datenbankseite. Ohne Ausbalancieren ist dies eine und mit sind dies zwei Seiten, d.h. die Komplexitätsklasse hierfür ist $O(1)$. Da es sich bei dem Subjektindex ebenfalls um einen B+-Baum handelt, betragen die Kosten für dessen Aktualisierung (Schritt 2) im schlechtesten Fall $O(\log_b M)$. Prädikat- und Objektindex sind gleich aufgebaut, weshalb die Schritte (3) und (4) gemeinsam betrachtet werden können. Sowohl beim Einfügen als auch beim Löschen müssen insgesamt zwei Bitsets (für Prädikat und Objekt) gelesen und geschrieben werden. Im schlechtesten Fall – alle Bits gesetzt – wären dies $\frac{2 \cdot 2N}{\|\mathrm{p}\|} + \frac{2 \cdot 2N}{\|\mathrm{p}\|}$ Lese- und Schreiboperationen, also bei fester Seitengröße $O(N)$ Operationen – im Allgemeinen wird dieser Fall bei RDF-Daten extrem selten auftreten.

Wenn man wiederum den Infobox-Datensatz zugrunde legt und, wie bereits im vorherigen Abschnitt dargestellt, davon ausgeht, dass die ersten beiden Ebenen von allen im Speichermodell vorhandenen B+-Bäumen im Hauptspeicher gehalten werden können und fast alle Bitsets nicht mehr als eine Seite belegen, ergibt sich für den günstigsten Fall (*engl. best case*) für das Einfügen oder Löschen eines Tripels folgende Rechnung:

(1)	Lesen von Subjektindex und Datenseite	2
(2)	Schreiben des Subjektindex	1
(3)	Lesen und Schreiben des Prädikatindex	2
(4)	Lesen und Schreiben des Objektindex	2
(5)	Schreiben der Datenseite	1
	Summe der Seitenoperationen	8

3.3.4 Vergleich mit anderen Speichermodellen

Die momentan am besten skalierenden Speichermodelle basieren auf dem Erzeugen von Indexen über unterschiedliche Kombinationen von Subjekt, Prädikat und Objekt (vgl. Abschnitt 3.1.2). Für das Bestimmen der Lösungen eines Basis-Graphmusters Q muss für jedes darin enthaltene Tripelmuster einmal ein Index angefragt und mit dem bisherigen Ergebnis verknüpft werden. Für das Verknüpfen der Lösungen für zwei Tripelmuster kann die Komplexität eines Index-Joins angenommen werden, bei dem auf beiden Eingabeströmen ein Index existiert. Entsprechend der relationalen Datenbanktheorie ist die Komplexität für diese Operation $O(N)$. Da die Daten in den betrachteten Speichermodellen sortiert gehalten werden, kann für die weiteren Verknüpfungen die Komplexität eines Sort-Merge-Joins angenommen werden, die ebenfalls mit $O(N)$ anzusetzen ist. Um letztendlich alle Tripelmuster miteinander zu verknüpfen, sind $|Q| - 1$ Join-Operationen erforderlich. Zusammengefasst haben auf Tripelindexen basierende Speichermodelle eine Komplexität von $O(|Q|N)$.

Das heißt, der Aufwand wächst bei gleicher Datenmenge proportional zu der Größe der Anfrage.

Bei Verwenden des RCS-Speichermodells (vgl. Abschnitt 3.2) steigt der Aufwand proportional zu der Anzahl der Konstanten Prädikate und Objekte und der sternförmigen Teilanfragen. Hierbei ist zu berücksichtigen, dass die Anzahl der Sternmuster im Allgemeinen wesentlich kleiner als die Anzahl der Tripelmuster in einer Anfrage ist.

3.4 Erweiterungen des RCS-Speichermodells

Das Einbeziehen von Informationen der gespeicherten Daten und des Anfrageprofils (*engl. query workload*) können Zugriffsstrukturen und Daten so gruppiert werden, dass die Anzahl der zugegriffenen Datenbankseiten gezielt reduziert werden kann. Zunächst wird ein geeignetes Gruppieren der Bitsets diskutiert und anschließend weitergehende Strategien zum Gruppieren von RDF-Tripeln auf den Datenbankseite angedacht.

3.4.1 Gruppieren von Zugriffsstrukturen

Bezüglich der Zugriffsstrukturen sind zwei Modifikationen denkbar: Zum einen können die Datenbankseiten bezüglich der Typen (`rdf:type`) der auf ihnen gespeicherten Subjekte gruppiert werden. Zum anderen können häufig gemeinsam angefragte Bitsets auf einer Seite gespeichert werden.

Das Gruppieren der Datenbankseiten bezüglich der Typinformation von Subjekten führt dazu, dass diese nahe liegende IDs. Dies hätte Einfluss auf die Kompressionsrate der Bitsets, da längere Läufe von Nullen bei Eigenschaften erzielt werden, deren Definitionsbereich nur wenige Klassen von Ressourcen umfasst. Da die Tripel im Sinne der Eigenschaften nun sortiert vorliegen, ließe sich eine Reduktion der Bitset-Größen um 30–40 % erzielen [LKA10]. Der Nachteil dieser Modifikation besteht in dem zusätzlichen Aufwand bei der Verwaltung der Datenbankseiten.

Das Zusammenfassen der Bitsets von häufig gemeinsam angefragten, selten vorkommenden Eigenschaften auf einer Seite würde dazu führen, dass bei einer Anfrage mit Konstanten für genau diese Eigenschaften weniger Seiten geladenen werden müssten. Damit und unter der Annahme, dass die ersten zwei Ebenen der B+-Bäume im Hauptspeicher gehalten werden, könnte die Komplexität der Seitenzugriffe im Idealfall (*engl. best case*) auf $O(|S|)$ reduziert werden und wäre damit proportional zu der Anzahl der Lösungen einer Anfrage.

Ein einfache Heuristik für das Zusammenfassen der Bitsets von Eigenschaften könnte auf den Definitions- und Wertebereichen dieser Eigenschaften basieren. Der zusätzliche Aufwand durch die vorgeschlagene Organisation ist als niedrig einzuschätzen, da dies als eine inhärente Charakteristik des B+-Baumes zur Verwaltung der Bitsets implementiert werden kann.

3.4.2 Gruppieren von Tripeln

Wie sich in Abschnitt 3.3 gezeigt hat, entstehen bei der Beantwortung von beliebigen, Basis-Graphmustern auch Kosten für das Verknüpfen der Lösungen

der sternförmigen Teilanfragen. Zwei Datenbankseiten sind miteinander implizit verknüpft, wenn ein Objekt auch als Subjekt verwendet wird (dieselbe Ressource-ID steht auf zwei Seiten). Eine Strategie für deren Reduktion könnte eine Optimierung dieser impliziten Verknüpfung sein, indem möglichst viele oder alle Objekte der einen Seite Subjekte auf der anderen sind (vgl. Abbildung 3.15).

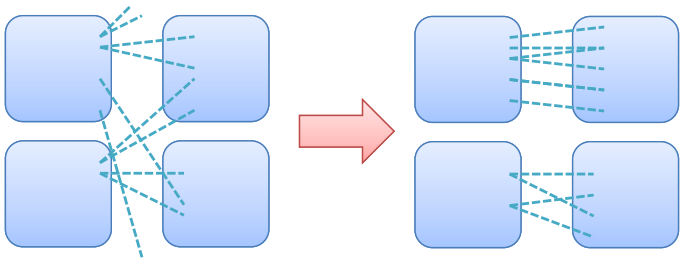


Abbildung 3.15: Optimierung von Verknüpfungen zwischen zwei Datenseiten

Insbesondere für oft gemeinsam in einer Anfrage auftretenden Klassen von Ressourcen erscheint eine derartige Gruppierung sinnvoll, da im Extremfall die Komplexität der Join-Operation von $O(nm)$ auf $O(n+m)$ reduziert werden kann.

Einen Schritt weiter geht die Überlegung diese häufig gemeinsam angefragten Klassen von Ressourcen auf derselben Seite zu speichern. Dies würde dem Konzept der schwachen Entitäten entsprechen, wie man sie aus der ER-Modellierung her kennt. Als Konsequenz würde eine Join-Operation eingespart werden. Als nachteilig könnte sich erweisen, dass nunmehr weniger Ressourcen eines Typs auf einer Seite gespeichert werden könnten, da auch Platz für die abhängigen Tripel benötigt wird.

3.5 Zusammenfassung

In diesem Kapitel wurde das neuartige Speichermodell des Ressource Centered Store (RCS) zur Verwaltung von RDF-Daten beschrieben. Im Gegensatz zur tripelorientierten Indexierung von existierenden nativen RDF-Datenbanksystemen werden im RCS die Tripel in Gruppen auf Datenseiten organisiert und lehnt sich damit an das gängige Speichermodell von relationalen DBMS an. Das wesentliche Merkmal einer Gruppe ist dabei das gemeinsame Subjekt. Aufgrund dieser Gruppierungseigenschaft lassen sich im Unterschied zu existierenden Ansätzen sternförmige Anfragen ohne das Ausführen von kostenintensiven Join-Operationen beantworten. Die Nähe des Speichermodells zu relationalen DBMS erlaubt ein Übertragen und Nachnutzen von Erkenntnissen über die Anfragebearbeitung in RDBMS auf den RCS (z. B. Join-Algorithmen).

Über den Datenseiten definierte Zugriffsstrukturen erlauben ein Lokalisieren der Datenseiten, die Informationen zu bestimmten RDF-Termen beinhalten. Darüber hinaus wird die Information, auf welchen Datenseiten sich ein Prädikat oder Objekt befindet, als Bitsets verwaltet und erlaubt damit durch Bitoperationen ein effizientes Bestimmen der Datenseiten auf den sich Tripel mit bestimmten RDF-Termen. Im Gegensatz zu den Indexstrukturen in rela-

tionalen DBMS kann dadurch die Menge der für eine Anfrage relevanten Datenseiten eingeschränkt werden. Die Verwendung von Bitsets als primäre Zugriffsstruktur stellt auch einen Unterschied zu den meisten existierenden nativen RDF-Datenbanksystemen dar.

Kapitel 4

Indexierung von RDF-Graphen

Neben einer geeigneten Verwaltung der RDF-Daten auf Datenbankseiten können zusätzliche Indexe genutzt werden, um ein effizientes Evaluieren von Anfragen zu gewährleisten. Im Jahr 1970 veröffentlichten Bayer und McCreight die Grundlagen für den B+-Baum [BM70]. Seitdem wird dieser als primäre Indexstruktur für das effiziente Lokalisieren von Einträgen auf Datenbankseiten verwendet. Während B-Bäume in relationalen DBMS eine zusätzliche Zugriffsmöglichkeit darstellen – neben dem sequentiellen Lesen aller zu einer Tabelle gehörenden Seiten –, werden diese in nativen RDF-DBMS oftmals auch zum Speichern der RDF-Tripel selbst herangezogen (vgl. Abschnitt 3.1). Um unabhängig von der Position der Variablen in einem Tripelmuster einen effizienten Zugriff auf die Tripel zu gewährleisten, werden mehrere B+-Bäume gleichzeitig eingesetzt, die verschiedene Kombinationen von Subjekt, Prädikat und Objekt indexieren. Da infolgedessen Tripel mehrfach gespeichert werden und keine feststehenden Adressen (wie Tupel-IDs in RDBMS) für Tripel existieren, ist ein Erweitern des Systems um weitere Indexstrukturen (z.B. Volltext-Index) aufwändiger.

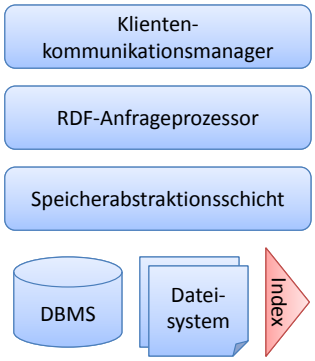


Abbildung 4.1: Generische Architektur eines RDF-Managementsystems

Im Resource-Centered Store hingegen wird der relationale Ansatz verfolgt und die Tripel auf Datenseiten gespeichert. Damit ist es möglich, jedem Tripel eine eindeutige Adresse bestehend aus Datenseite und dessen Position auf dieser zuzuordnen. Aufgrund der Verwaltung auf Datenseiten können mit wenig Aufwand weitere spezialisierte Zugriffsstukturen in das System integriert werden (vgl. hervorgehobene Komponente in Abbildung 4.1).

Ohne zusätzliche Indexe können im RCS vor allem sternförmige Anfragen effizient beantwortet werden, da alle Tripel mit demselben Subjekt auf einer

Datenbankseite zu finden sind. Anfragen mit einer komplexeren Struktur – bestehend aus mehreren sternförmigen Muster – werden zunächst in sternförmige Teilanfragen zerlegt. Diese werden dann mit Hilfe der Subjekt-, Prädikat-, und Objektindexe ausgewertet und die entstehenden Ergebnisse miteinander verknüpft (Join-Operationen). In diesem Kapitel werden nutzerdefinierte Indexe über größere Graphmuster besprochen, mit denen häufig auftretenden Join-Operationen vorberechnet werden können und somit die Anfrageausführung beschleunigen. Während in native RDF-Datenbanksysteme nur festgelegte und automatisch generierte Indexe existieren, können im Ressource Centered Store beliebige Graphmuster indexiert werden. Genau wie in RDBMS mehrere Indexe über dieselbe Spalte erstellt werden können, können im RCS mehrere Indexe über demselben Teilmuster unabhängig voneinander existieren. Der Fokus dieses Kapitels liegt dabei auf der Indexierung von Graphmustern und nicht auf dem Indexieren von Ressourcen und Literalen. Für letztere werden existierende Verfahren eingesetzt (z. B. [CR02]).

Im Abschnitt 4.1 wird zunächst die Problemstellung formalisiert und in Teilprobleme gegliedert. Um für die Beantwortung einer Anfrage ein oder mehrere Indexe verwenden zu können, werden in Abschnitt 4.2 die Beziehungen zwischen Graphmustern bzw. zwischen Graphmuster und Lösungen untersucht. Die daraus gewonnenen Kenntnisse werden in Abschnitt 4.3 genutzt, um geeignete Indexe zur Anfragebearbeitung auszuwählen. Im Anschluss daran wird die Verwaltung von Indexen im RCS beschrieben (Abschnitt 4.4) und ein Verfahren zur Berechnung der Kosten für einen Indexzugriff vorgestellt (Abschnitt 4.5). Abschließend werden im Abschnitt 4.6 verwandte Arbeiten betrachtet und in Abschnitt 4.7 die Ergebnisse dieses Kapitels zusammengefasst.

4.1 Problemdefinition

Das Ziel des Indexierens von RDF-Daten ist das schnelle Auffinden von darin enthaltenen Teilgraphen, die einem gegebenen Graphmustern entsprechen. Der in dieser Arbeit verfolgte Ansatz geht von einer Menge von auf Graphmustern basierenden Indexen aus, aus denen ein oder mehrere zur effizienten Beantwortung während der Anfrageverarbeitung ausgewählt werden. Zunächst werden in diesem Abschnitt grundlegende Begriffe eingeführt, um darauf aufbauend die Problemstellung zu formulieren.

Die Grundlage für die Definition von Indexen über RDF-Graphen bilden aus einer Menge von Tripelmustern bestehende Basis-Graphmuster, wie sie in der SPARQL-Spezifikation [PS13] beschrieben sind (vgl. Definitionen 2.5 und 2.6 auf Seite 10). Im Folgenden wird ein Basis-Graphmuster, auf dem ein Index über einem RDF-Graphen basiert, kurz als *Indexmuster* bezeichnet.

SPARQL ermöglicht zwar auch das Formulieren von komplexeren Anfragen, indem beispielsweise Basis-Graphmuster miteinander in Beziehung gesetzt werden (z. B. durch `UNION` oder `OPTIONAL`) oder die Ergebnismenge durch das Verwenden von so genannten Lösungsmengenmodifikatoren (z. B. Projektion, `LIMIT`) eingeschränkt wird. Diese basieren jedoch auf Basis-Graphmustern und werden somit als Spezialfälle hier nicht gesondert betrachtet.

Die Einträge in einem Index bilden alle Lösungen (vgl. Definition 2.9 auf Seite 12) des zugrunde liegenden Indexmusters über einen gegebenen RDF-

Graphen. Demzufolge kann dieselbe Lösung auch mit einer höheren Kardinalität im Index auftreten.

Definition 4.1 (Index)**.** *Sei $G \in \mathcal{G}$ ein beliebiger RDF-Graph. Ein* Index *I über G ist ein Paar $I = (P, \Omega_P)$ mit $P \in \mathsf{BGP}$ und $\Omega_P = \{\mu_1, \ldots, \mu_n\}$ ist die Multimenge der Lösungen für P über G. Die Größe eines Indexes wird als $|I| = |\Omega_P|$ notiert.*

Betrachtet man die Definition für das Bestimmen der Lösungen für ein Basis-Graphmuster, so verhalten sich anonyme Ressourcen wie Variablen: Anonyme Ressourcen werden mit Hilfe einer RDF-Instanz-Abbildung durch RDF-Terme ersetzt. In diesem Sinne wären anonyme Ressourcen in einem Indexmuster nur dann sinnvoll, wenn in einem Index deren Bindungen ignoriert werden sollen. Falls nicht gesondert angemerkt, wird deswegen im Folgenden angenommen, dass ein Indexmuster keine anonyme Ressourcen beinhaltet. Darüber hinaus sei angemerkt, dass die obige Definition eines Index ohne Beschränkung der Allgemeinheit nur über einen einzelnen RDF-Graphen definiert ist. Wenn ein Index über mehrere RDF-Graphen betrachtet werden soll, so würde man zunächst die Vereinigung der RDF-Graphen bilden (vgl. [PS13]) und anschließend die Lösungen für das Indexmuster bestimmen. Ein solcher Index könnte dann nur für Anfragen verwendet werden, die über die jeweiligen RDF-Graphen definiert ist.

Grundsätzlich können mehrere Indexe über einem RDF-Graphen definiert sein. Genauso wie in relationalen DBMS ergibt sich daraus beim Generieren der möglichen Ausführungspläne für eine Anfrage über diesen RDF-Graphen die Fragestellung, ob die Verwendung von ein oder mehreren Indexe im Anfrageplan die Kosten der Anfrageausführung reduzieren können. Die Kosten können mittels einer Kostenfunktion abgeschätzt werden. Diese Fragestellung wird in dem folgenden Hauptproblem formalisiert:

Problem 4.1 (Hauptproblem)**.** *Gegeben sei eine Menge von Indexen $\mathcal{I} = \{I_1 \ldots I_n\}$, eine Anfrage Q über einem RDF-Graphen $G \in \mathcal{G}$ sowie eine Kostenfunktion c. Gesucht ist eine Menge $I \subseteq \mathcal{I}$ mit $c(Q, I)$ minimal.*

Aus der obigen, generisch formulierten Problemstellung lassen sich die beiden Teilprobleme *Indexauswahl* und *Kostenfunktion* ableiten. Das erste Teilproblem betrifft die Frage, welche Indexe überhaupt Ergebnisse zu der Beantwortung einer Anfrage beitragen und somit sinnvollerweise bei der Erstellung von Anfrageplänen in Betracht gezogen werden können.

Problem 4.2 (Indexauswahl)**.** *Gegeben sei eine Menge von Indexen $\mathcal{I}$ und eine Anfrage Q über einem RDF-Graphen $G \in \mathcal{G}$. Gesucht ist die Menge von $I \subseteq \mathcal{I}$, die Teillösungen zur Evaluation von Q beitragen können.*

Das zweite beschäftigt sich mit der Kostenfunktion. Falls mehrere Indexe potentiell für die Beantwortung einer Anfrage genutzt werden können, muss eine geeignete Auswahl getroffen werden. Dazu ist ein Abschätzen der Kosten für das Ausführen der Anfragen ohne Index bzw. unter Verwendung von ein oder mehreren Indexen erforderlich. Wenn Indexe bei der Anfrageausführung verwendet werden, dann zerfällt eine Anfrage Q in einen von den Indexmustern überdeckten Teil Q_I und einen Rest Q_R. Die Gesamtkosten ergeben sich somit zum einen aus den Kosten für das Ermitteln der Teillösungen für Q_I und für Q_R und zum anderen aus den Kosten für das Vervollständigen der Teillösungen zu einer Lösung für die Anfrage Q.

Problem 4.3 (Kostenfunktion)**.** *Gegeben sei eine Menge von Indexen $I \subseteq \mathcal{I}$ und eine Anfrage Q über einen RDF-Graphen $G \in \mathcal{G}$, wobei die Indexe in I Teillösungen zu Q beitragen können. Gesucht ist eine Kostenfunktion $c(Q,I) = c_{Q_I} + c_{Q_R} + c_J$, wobei c_{Q_I}, c_{Q_R} die Kosten der Berechnung der Teillösungen und c_J die Kosten der Vervollständigung zu Lösungen repräsentieren.*

In den folgenden Abschnitten wird insbesondere der Aspekt betrachtet, dass mindestens ein Index Teillösungen zu der der Gesamtlösung einer Anfrage beitragen kann. Wenn kein Indexmuster das Anfragemuster überdeckt, so zerfällt die Anfrage nicht in Q_I und Q_r, sondern es gilt $Q_R = Q$ und $c(Q,I) = c_{Q_R} = c_Q$. Wenn hingegen ein Indexmuster exakt dem Anfragemuster entspricht, dann zerfällt das Anfragemuster ebenfalls nicht und wir erhalten $Q_I = Q$ und $c(Q,I) = c_{Q_I} = c_Q$.

Damit ein Index wie in Problem 4.2 zu der Berechnung der Lösungen einer Anfrage beitragen kann, müssen Anfrage und Indexmuster Gemeinsamkeiten aufweisen, z.B. teilweise dieselben Ressourcen und/oder Eigenschaften referenzieren. In diesem Abschnitt werden die notwendigen und hinreichenden Bedingungen für die Verwendbarkeit eines Indexes zur Beantwortung einer Anfrage dargestellt.

Ausgehend von dem Gedanken, dass Anfrage- und Indexmuster Gemeinsamkeiten aufweisen müssen, wird zunächst definiert, wann sich zwei Graphmuster überlappen bzw. sie ineinander enthalten sind. Da ein Indexmuster aus einem Basis-Graphmuster (vgl. Definition 4.1) und eine Anfrage aus miteinander verknüpften Basis-Graphmustern (vgl. Definition 2.6) besteht, lassen sich aus dem Betrachten der Beziehungen zwischen Basis-Graphmustern alle anderen Konstellationen herleiten. Daher werden in diesem Abschnitt zunächst die Beziehungen zwischen zwei Basis-Graphmuster betrachtet.

Die gewonnen Erkenntnisse werden auf die Beziehung zwischen Index- und Anfragemuster übertragen und analysiert, inwiefern die Einträge eines Indexes zur Berechnung der Lösungen einer Anfrage herangezogen werden können. Im Anschluss daran wird auf Beziehungen zwischen mehreren Graphmustern eingegangen, um den Fall von mehreren Indexen und einem Anfragemuster beurteilen zu können.

4.2 Basis-Graphmuster und Lösungen

In diesem Abschnitt werden die Voraussetzungen geschaffen, um im weiteren Verlauf entscheiden zu können, ob zum einen ein Index potentiell zur Beantwortung einer Anfrage eingesetzt werden kann und zum anderen mehrere Indexe gleichzeitig verwendet werden können. Der erste Punkt tangiert die Frage, wann die in einem Index gespeicherten Lösungen eine Teillösung für ein Graphmuster bilden. Der zweite Punkt beschäftigt sich hingegen mit der Frage, in welcher Beziehung die Graphmuster zweier Indexe zueinander stehen müssen, damit die Lösungen der beiden Indexe zu einer Teillösung der Anfrage verknüpft werden können.

4.2.1 Beziehungen zwischen Basis-Graphmustern

Die Frage, ob ein Basis-Graphmuster in einem anderen vorkommt oder sie sich überlappen, ist intuitiv leicht verständlich: Im ersteren Fall ist das eine

Graphmuster ein Teilmuster des anderen und im letzteren besitzen die beiden Graphmuster ein gemeinsames Teilmuster. Um diese Sachverhalte zu formalisieren, wird zunächst eine Abbildung definiert, mit der einer Variablen nicht nur ein RDF-Term sondern auch eine Variable zugeordnet werden kann. Mit einer solchen Abbildung können beispielsweise die Variablen eines Graphmusters so umbenannt werden, dass es in ein anderes überführt werden kann.

Definition 4.2 (Variablenabbildung)**.** *Eine* Variablenabbildung ν *ist eine partielle Funktion* $\nu : \mathbb{V} \rightarrow \mathsf{RDF\text{-}T} \cup \mathbb{V}$. *Der Definitionsbereich von* ν *ist die Teilmenge von* $\mathbb{V}$, *auf der* ν *definiert ist.*

Eine Variablenabbildung ν kann sowohl auf eine Lösungsabbildung μ als auch auf ein Graphmuster P angewendet werden. Wird eine Variablenabbildung auf eine Lösungsabbildung (vgl. Definition 2.7 auf Seite 11) angewendet, so wird eine neue Lösungsabbildung konstruiert, deren Definitionsbereich der Bildmenge der Variablenabbildung entspricht. Die entstehende Lösungsabbildung ist wie folgt definiert:

Definition 4.3. *Das Anwenden einer Variablenabbildung* ν *auf eine Lösungsabbildung* μ, *notiert als* $\nu[\mu]$, *resultiert in eine neue Lösungsabbildung* μ'*:*

$$\mu'(x) = \begin{cases} \mu(y) & \textit{falls } \exists y \in \mathrm{dom}(\nu) : y \in \mathrm{dom}(\mu) \wedge \nu(y) = x \wedge \\ & \forall y' \in \mathrm{dom}(\nu) : \nu(y') = x \Rightarrow \mu(y') = \mu(y) \\ \textit{undef.} & \textit{sonst} \end{cases}$$

Der Definitionsbereich von μ', *notiert als* $\mathrm{dom}(\mu')$, *ist gleich der Bildmenge von* ν, *notiert als* $\mathrm{ran}(\nu)$.

Die komplexe Bedingung in der obigen Definition stellt sicher, dass jedem $x \in \mathrm{dom}(\mu')$ auch dann genau ein Wert zugeordnet wird, wenn die Abbildung ν zwei unterschiedliche Variablen auf dieselbe abbildet.

Das Anwenden einer Variablenabbildung auf ein Basis-Graphmuster wird durch die folgende Definition beschrieben:

Definition 4.4. *Das Anwenden einer Variablenabbildung* ν *auf ein Graphmuster* $P \in \mathsf{BGP}$, *notiert als* $\nu[P]$, *resultiert in ein neues Graphmuster* P', *in dem jede Variable* $x \in \mathrm{dom}(\nu)$ *durch* $\nu(x)$ *ersetzt ist.*

Damit ist die Definition einer Variablenabbildung eine Verallgemeinerung der Definition einer Lösungsabbildung (vgl. Definition2.7). Eine Variablenabbildung ν ist genau dann eine Lösungsabbildung, wenn die Bildmenge von ν keine Variablen enthält. Wenn im Gegensatz dazu die Bildmenge $\mathrm{ran}(\nu)$ nur Variablen enthält, so findet eine Umbenennung der Variablen statt. In diesem Fall wird im Folgenden von einer *Variablenumbenennung* gesprochen. Kombiniert man eine Variablenbindung mit einer RDF-Instanzabbildung (vgl. Definition 2.8 auf Seite 11), so können in einem Basis-Graphmuster Variablen umbenannt bzw. anonyme Ressourcen ersetzt werden.

Definition 4.5 (Graphmusterabbildung)**.** *Eine* Graphmusterabbildung γ *ist die Kombination einer RDF-Instanzabbildung* σ *und einer Variablenabbildung* ν. $\gamma[P] = \nu[\sigma[P]] = \nu \circ \sigma[P]$ *mit* $P \in \mathsf{BGP}$.

Mit Hilfe der beiden obigen Definitionen lässt sich nun feststellen, ob ein Graphmuster in einem anderen enthalten ist. Dazu wird nach der Existenz einer Graphmusterabbildung $\gamma = \nu \circ \sigma[P]$ gefragt, so dass das transformierte Graphmuster ein Teilgraph des anderen ist. Hierbei können zu einem gegebenen Graphmuster mehrere mögliche Variablenabbildungen ν existieren.

Definition 4.6 (Enthaltensein)**.** *Seien $P_1, P_2 \in \mathsf{BGP}$ zwei Basis-Graphmuster. Das Graphmuster P_2* enthält *P_1, notiert als $P_1 \sqsubseteq P_2$, wenn es eine Graphmusterabbildung $\gamma = \nu \circ \sigma$ gibt, so dass $\gamma[P_1]$ ein Teilgraph von P_2 ist.*

$$P_1 \sqsubseteq P_2 :\Leftrightarrow \exists \gamma : \gamma[P_1] \subseteq P_2$$

Die verschiedenen Variablenabbildungen $\nu_1 \ldots \nu_n$ für die P_1 in P_2 enthalten ist, werden als Vorkommen *von P_1 in P_2 bezeichnet.*

Abbildung 4.2 illustriert das Anwenden von Definition 4.6 anhand eines Beispiels. Die Basis-Graphmuster P_1 und P_2 beschreiben dieselben Teilgraphen in einem RDF-Graphen. Die beiden Graphmuster beschreiben Ressourcen, denen ein Name und ein aktuelles Projekt zugeordnet worden sind. Jedoch hat das zweite Graphmuster in der Subjektposition anstelle einer Variablen eine anonyme Ressource. In dem Beispiel enthält das Graphmuster P_3 das Graphmuster P_1, aber nicht P_2. Letzteres ist nicht in P_3 enthalten, da anonyme Ressourcen mittels einer RDF-Instanzabbildung nur auf RDF-Terme abgebildet werden können und es somit keine Abbildung gibt, die P_2 in einen Teilgraphen von P_3 transformiert.

```
P1 : { ?person foaf:currentProject ?project .
       ?person foaf:name ?name . }

P2 : { _:p foaf:currentProject ?project .
       _:p foaf:name ?name . }

P3 : { ?person foaf:currentProject ex:dbpedia .
       ?person foaf:name ?name .
       ?person foaf:mbox ?mail . }
```

Abbildung 4.2: Basis-Graphmuster P_3 enthält das Graphmuster P_1 mit ν: {?project → ex:dbpedia}, aber nicht P_2

Unmittelbar aus der Definition 4.6 lässt sich ableiten, dass das Anwenden einer Graphmusterabbildung nur Graphmuster erzeugen kann, die das ursprüngliche Graphmuster enthalten:

Lemma 4.1. *Sei mit $P \in \mathsf{BGP}$ ein Basis-Graphmuster gegeben. Dann gilt die folgende Aussage:*

$$\gamma \text{ ist Graphmusterabbildung } \Rightarrow P \sqsubseteq \gamma[P]$$

Beweis. Es gilt eine Graphmusterabbildung γ' zu finden, so dass $\gamma'[P] \subseteq \gamma[P]$ gilt. Man wähle $\gamma' = \gamma$. □

Aufbauend auf dem Begriff des Enthaltenseins lässt sich die Äquivalenz zweier Basis-Graphmuster definieren.

Definition 4.7 (Äquivalenz)**.** *Die Graphmuster sind genau dann* äquivalent, $P_1 \sim P_2$*, wenn sie sich gegenseitig enthalten.*

$$P_1 \sim P_2 :\Leftrightarrow P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$$

Wenn ein Graphmuster nicht in einem anderen komplett enthalten ist, so besteht noch die Möglichkeit, dass diese sich überlappen. In diesem Fall würde es einen gemeinsamen Teilgraphen geben, der in beiden Graphmustern enthalten ist. Wie sich anhand der Beispiele in Abbildung 4.3 zeigen lässt, gibt es zwei Arten der Überlappung. Während P_1 und P_2 ein gemeinsames Tripelmuster beinhalten und sich somit gegenseitig (beidseitig) überlappen, überlappt P_1 das Graphmuster P_3 nur insofern, als dass erste Tripelmuster von P_1 in P_3 enthalten ist (einseitig).

```
P1 : { ?person foaf:currentProject ?project .
       ?person foaf:homepage ?hp . }

P2 : { ?person foaf:currentProject ?project .
       ?person foaf:interest ?document . }

P3 : { ?person foaf:currentProject ex:dbpedia .
       ?person foaf:name ?name .
       ?person foaf:mbox ?mail . }
```

Abbildung 4.3: Basis-Graphmuster P_1 und P_2 überlappen sich beidseitig, während P_1 und P_3 mit ν: {?project → ex:dbpedia} sich nur einseitig überlappen

Die nachfolgende Definition beschreibt das einseitige Überlappen zweier Basis-Graphmuster:

Definition 4.8 (Einseitiges Überlappen)**.** *Seien* $P_1, P_2 \in \mathsf{BGP}$ *zwei Basis-Graphmuster. Das Graphmuster* P_1 überlappt einseitig P_2*, notiert als* $P_1 \bullet\!\circ P_2$*, wenn ein Teilgraph* $P \subset P_1$ *in* P_2 *enthalten ist.*

$$P_1 \bullet\!\circ P_2 :\Leftrightarrow \exists P \subset P_1 : P \sqsubset P_2$$

Beim Definieren des beidseitigen Überlappens ist es nicht hinreichend zu fordern, dass sich die Graphmuster jeweils einseitig überlappen (vgl. Abbildung 4.4). Darüber hinaus muss gesichert werden, dass sich die Graphmuster mit zueinander äquivalenten Teilgraphen überlappen. Somit ergibt sich die folgende Definition:

Definition 4.9 (Beidseitiges Überlappen)**.** *Seien* $P_1, P_2 \in \mathsf{BGP}$ *zwei Basis-Graphmuster. Die Graphmuster* P_1 *und* P_2 überlappen sich gegenseitig*, notiert als* $P_1 \bullet\!\bullet P_2$*, wenn es Teilgraphen* $P \subset P_1$ *und* $P' \subset P_2$ *gibt, so dass* P *äquivalent zu* P' *sowie* P' *in* P_1 *und* P *in* P_2 *enthalten ist.*

$$P_1 \bullet\!\bullet P_2 :\Leftrightarrow \exists P \subset P_1 \exists P' \subset P_2 : P \sqsubset P_2 \wedge P' \sqsubset P_1 \wedge P \sim P'$$

Obwohl die Vereinigung zweier Basis-Graphmuster, $P_1 \cup P_2$, auf den ersten Blick keine Beziehung zwischen diesen darstellt, bedarf es dennoch einer ge-

```
P1 : { ?person foaf:currentProject ex:dbpedia .
       ?person foaf:interest ?document . }

P2 : { ?person foaf:currentProject ?project .
       ?person foaf:interest ex:einBericht . }
```

Abbildung 4.4: Basis-Graphmuster P_1 und P_2 überlappen sich jeweils einseitig (ν_1: {?project → ex:dbpedia}, ν_2: {?document → ex:einBericht}), aber nicht beidseitig

naueren Betrachtung. Die in Abbildung 4.5 gezeigten Graphmuster verdeutlichen die Problematik. Ohne weitere Informationen über die Beziehungen zwischen den Variablen der Graphmuster ist die Vereinigung $P_1 \cup P_2$ nicht eindeutig. Im Beispiel wären mindestens drei Variationen möglich, die die Lösungsmengen für die jeweilige Vereinigung stark beeinflussen. Die Variationen umfassen: (i) ?person1 = ?person, (ii) ?person2 = ?person und (iii) ?person ≠ ?person1 ∧ ?person1 ≠ ?person2.

```
P1 : { ?person1 foaf:knows ?person2 . }

P2 : { ?person foaf:interest ex:einBericht . }
```

Abbildung 4.5: Vereinigung zweier Basis-Graphmuster P_1 und P_2 kann mehrdeutig sein

Für das Bilden der Vereinigung von zwei Graphmustern sind also die verwendeten Variablenabbildungen entscheidend. Daraus leitet sich die folgende Definition für die Vereinigung ab:

Definition 4.10 (Vereinigung)**.** *Seien mit $P_1, P_2 \in \mathsf{BGP}$ zwei Basis-Graphmuster gegeben. Die Vereinigung von P_1 und P_2, notiert als $P_1 \sqcup P_2$, ist definiert als $\gamma_1[P_1] \cup \gamma_2[P_2]$ für zwei beliebige Graphmusterabbildungen γ_1 und γ_2.*

Unmittelbar aus obiger Definition ist ersichtlich, dass die jeweiligen Ausgangsgraphmuster in der Vereinigung enthalten sein muss. Somit gilt folgendes Lemma:

Lemma 4.2. *Seien mit $P_1, P_2 \in \mathsf{BGP}$ zwei Graphmuster gegeben. Es gilt die folgende Aussage:*

$$P_1 \sqcup P_2 \Rightarrow P_1 \sqsubseteq P_1 \sqcup P_2 \wedge P_2 \sqsubseteq P_1 \sqcup P_2$$

Beweis. per definitionem. □

In Definition 4.10 werden die anzuwendenden Graphmusterabbildungen nicht näher spezifiziert und als gegeben vorausgesetzt. Mit Blick auf das Problem der Indexauswahl 4.2 lässt sich die Menge der Graphmusterabbildungen dadurch einschränken, dass ein drittes Graphmuster P als Ziel vorgegeben wird, in das die beiden Graphmuster eingebettet werden.

Definition 4.11. *Seien mit $P_1, P_2, P \in \mathsf{BGP}$ drei Basis-Graphmuster gegeben. Für zwei Graphmusterabbildungen γ_1 und γ_2 mit $\gamma_1[P_1] \subseteq P$ und $\gamma_2[P_2] \subseteq P$ bezeichnet $P_1 \sqcup_P P_2 = \gamma_1[P_1] \cup \gamma_2[P_2]$ die Vereinigung von P_1 und P_2 bezüglich eines Graphmusters P.*

Im Allgemeinen ist die Wahl der Graphmusterabbildungen nicht eindeutig, jedoch hilft die Definition bestimmte Abbildungen auszuschließen. Zum Beispiel ist die Vereinigung von P_1 und P_2 aus Abbildung 4.5 bezüglich des folgenden Graphmusters eindeutig ($\gamma_1 = \{?\texttt{person1} \to ?\texttt{p}, ?\texttt{person2} \to ?\texttt{q}\}$ und $\gamma_2 = \{?\texttt{person} \to ?\texttt{p}\}$):

```
P: { ?p foaf:knows ?q .
     ?p foaf:interest ?document .
     ?p foaf:name "Jeff" . }
```

4.2.2 Zusammenhänge zwischen Graphmustern und ihren Lösungen

Als nächsten Schritt zum Annähern an das Problem 4.2 wird in diesem Abschnitt untersucht, welche Rückschlüsse aus den Beziehungen zwischen zwei Basis-Graphmustern auf die Verwertbarkeit deren Lösungen bezüglich eines RDF-Graphen G gezogen werden können. Konkret werden im Folgenden die nachfolgenden Fragestellungen betrachtet:

Problem 4.4. *Seien zwei zueinander in Beziehung ($\sqsubseteq, \bullet\bullet, \bullet\circ$) stehende Basis-Graphmuster $P_1 \in \mathsf{BGP}$ und $P_2 \in \mathsf{BGP}$ sowie ein RDF-Graph $G \in \mathcal{G}$ gegeben.*

1. *Wie wirken sich Variablen- und RDF-Instanzabbildungen auf die Lösungen aus?*

2. *Wie stehen die Lösungen für P_1 über G zu den Lösungen von P_2 über G in Beziehung?*

3. *Wie stehen die Lösungen für P_1 und P_2 über G zu den Lösungen von $P_1 \sqcup P_2$ über G in Beziehung?*

Der ersten beiden Fragestellungen sind im vorliegenden Kontext bedeutsam, da damit entschieden werden kann, ob und welche Indexe für die Evaluation einer Anfrage überhaupt herangezogen werden können. Die Erkenntnisse zur dritten helfen zu entscheiden, inwiefern mehrere Indexe gleichzeitig genutzt werden können und wie aus jeweils einer Lösung dieser Indexe eine Teillösung einer Anfrage konstruiert werden kann.

Da die in Problem 4.4 genannten Beziehungen zwischen Basis-Graphmustern auf Graphmusterabbildungen und diese wiederum auf die Hintereinanderausführung von RDF-Instanz- und Variablenabbildungen basieren, werden zunächst die letzteren beiden Abbildungen untersucht. Die gewonnenen Erkenntnisse können unmittelbar auf Graphmusterabbildungen übertragen werden, da die Definitionsbereiche der beiden Abbildungen disjunkt sind.

4.2.2.1 Lösungen einer RDF-Instanzabbildung bzgl. eines Graphmusters: $\sigma[P]$

In diesem Abschnitt ist der Ausgangspunkt, dass eine Lösung μ für ein Basis-Graphmuster P bekannt ist. Es wäre wünschenswert, wenn nach dem Anwenden von Variablen- und RDF-Instanzabbildungen aus der bekannten Lösung eine für das modifizierte Graphmuster abgeleitet werden kann. Die beiden Abbildungen werden im Folgenden nacheinander betrachtet.

Sei zuerst eine RDF-Instanzabbildung σ gegeben, mit der in einem Basis-Graphmuster P anonyme Ressourcen entweder durch anonyme Ressourcen, Ressource-IRIs oder Literale ersetzt werden. Wie das Gegenbeispiel in Abbildung 4.6 veranschaulicht, kann im Allgemeinen nicht davon ausgegangen werden, dass aus einer Lösung für ein Graphmuster P eine Lösung für $\sigma[P]$ abgeleitet werden kann. Während es im Beispiel für P eine Lösung über den Graphen G gibt, existiert keine Lösung für $\sigma[P]$ mit $\sigma(\_:\mathtt{p}) = \mathtt{ex:einProjekt}$. Das Problem darin zu suchen, dass σ eine anonyme Ressource auf eine Ressource-IRI bzw. ein Literal abbildet und dadurch $\sigma[P]$ für keine Lösung $\mu \in \Omega_P$ mehr ein Teilgraph von G sein kann.

G: `{ ex:einePerson foaf:currentProject ex:dbpedia . }`

P: `{ ?person foaf:currentProject _:p . }`

$\sigma[P]$: `{ ?person foaf:currentProject ex:einProjekt . }`

Abbildung 4.6: Im Allgemeinen gilt: $\mu \in \Omega_P \nRightarrow \mu \in \Omega_{\nu[P]}$

Ein Gegenbeispiel kann auch für eine Abbildung konstruiert werden, bei der zwei anonyme Ressourcen aus P auf dieselbe anonyme Ressource oder eine anonyme Resource auf eine anonyme Ressource von P abgebildet werden. In diesem Fall sind nur die $\mu \in \Omega_P$ ein Lösung, bei der diese beiden Ressourcen mit demselben Wert belegt werden, d.h. im Allgemeinen gibt es weniger Lösungen für $\sigma[P]$.

Lemma 4.3. *Sei $P \in \mathsf{BGP}$, ein RDF-Graph $G \in \mathcal{G}$ und eine RDF-Instanzabbildung σ gegeben. Die Aussage $\mu \in \Omega_P \Leftrightarrow \mu \in \Omega_{\sigma[P]}$ gilt nur dann, wenn σ die folgenden Eigenschaften erfüllt:*

i) $\mathrm{ran}(\sigma) \subseteq \mathbb{B}$

ii) $\mathrm{ran}(\sigma)$ *und die Menge anonymer Ressourcen von P sind disjunkt*

iii) $\forall x, y \in \mathrm{dom}(\sigma) : x \neq y \rightarrow \sigma(x) \neq \sigma(y)$

Beweis. Die Aussage gilt offensichtlich, da durch σ nur eine Umbenennung der anonymen Ressourcen stattfindet. □

Das Lemma 4.3 schränkt die Menge der RDF-Instanzabbildungen ein, für die man aus einer Lösung für P unmittelbar eine für $\sigma[P]$ bestimmen kann. Betrachtet man jedoch die entgegen gesetzt Richtung der Implikation, so kann eine stärkere Aussage getroffen werden.

Lemma 4.4. *Sei ein Graphmuster $P \in \mathsf{BGP}$, ein RDF-Graph $G \in \mathcal{G}$ und eine RDF-Instanzabbildung σ gegeben. Es gilt die folgende Aussage: $\mu \in \Omega_{\sigma[P]} \Rightarrow \mu \in \Omega_P$*

Beweis. Sei $\mu \in \Omega_{\sigma[P]}$ gegeben. Laut Definition 2.9 ist $\mu \in \Omega_P$, wenn es eine RDF-Instanzabbildung σ gibt, so dass $\mu[\sigma[P]] \subset G$. □

Zusammenfassend lässt sich also feststellen, dass im Allgemeinen die Lösungen eines Basis-Graphmusters P nur Kandidaten für Lösungen von $\sigma[P]$ darstellen. Nur wenn eine RDF-Instanzabbildung die in Lemma 4.3 geforderten Eigenschaften erfüllt, können diese ohne weitere Überprüfung als Lösung für $\sigma[P]$ betrachtet werden. Der Grund liegt in der Tatsache, dass in einer Lösungsabbildung keinerlei Informationen über die Belegung der anonymen Ressourcen enthalten sind. Insbesondere ergibt sich daraus auch, dass σ keine Auswirkungen auf eine Lösungsabbildung μ hat. Nicht zuletzt zeigt das Lemma auch, dass es unter den genannten Bedingungen keine weiteren Lösungen für $\sigma[P]$ geben kann, d.h. $\Omega_P = \Omega_{\sigma[P]}$. Aus Lemma 4.4 lässt sich schließlich $\Omega_{\sigma[P]} \subseteq \Omega_P$ ableiten.

4.2.2.2 Lösungen einer Variablenabbildung bzgl. eines Graphmusters: $\nu[P]$

Als nächstes sei eine Variablenabbildung ν gegeben. Um alle Facetten der Frage der Übertragbarkeit von Lösungen von P auf $\nu[P]$ zu beleuchten, werden die folgenden Möglichkeiten der Abbildung einer Variablen gesondert untersucht: (1) eine Variable, (2) eine anonyme Ressource oder (3) eine Ressource-IRI bzw. Literal.

Zunächst wird der Fall betrachtet, in dem Variablen nur auf Variablen abgebildet werden. Falls mittels ν nur eine Umbenennung von Variablen vorgenommen wird, es gilt also $\forall x, y \in \mathrm{dom}(\nu) : x \neq y \Rightarrow \nu(x) \neq \nu(y)$, sind offensichtlich die Lösungen eines Musters P auch Lösungen von $\nu[P]$, indem man die Umbenennung auch auf die Lösungen anwendet. Die Situation verändert sich erst, wenn durch ν zwei Variablen von P auf dieselbe Variable abgebildet werden ($x \neq y$ aber $\nu(x) = \nu(y)$). Um aus einer Lösung von P eine Lösung für $\nu(P)$ herzuleiten, dürfen nur Lösungen $\mu \in \Omega_P$ mit $\mu(x) = \mu(y)$ betrachtet werden. Der letztere Fall wird durch Abbildung 4.7 illustriert. Während für P zwei Lösungen über G existieren, gibt es beim Graphmuster $\nu[P]$ mit $\nu = \{?person1 \rightarrow ?person, ?person2 \rightarrow ?person\}$ nur noch eine. Diese lässt sich jedoch aus den Lösungen für P herleiten.

G:
```
{ ex:einePerson foaf:knows ex:zweitePerson .
  ex:einePerson foaf:knows ex:einePerson . }
```

P:
```
{ ?person1 foaf:knows ?person2 . }
```

$\nu[P]$:
```
{ ?person foaf:knows ?person . }
```

Abbildung 4.7: Ω_P enthält mehr Lösungen als $\Omega_{\nu[P]}$

Der zweite Fall, die Abbildung auf eine anonyme Ressource, kann analog zur Abbildung auf eine Variable betrachtet werden, da sich bei der Auswertung einer Anfrage anonyme Ressourcen ähnlich wie Variablen verhalten; je-

doch kann einer Lösung nicht entnommen, an welche RDF-Terme eine anonyme Ressource gebunden worden ist. Insbesondere sind die Kardinalitäten der jeweiligen Lösungsmengen identisch (vgl. Abschnitt 2.2, Seite 11). Wenn beispielsweise in Abbildung 4.7 die Variable ?person2 durch ν auf eine anonyme Ressource abgebildet werden würde, dann würde die Lösungsmenge noch immer zwei Elemente enthalten.

Zuletzt wird der Fall untersucht, bei dem durch die Variablenabbildung eine Variable auf eine Ressource-IRI oder ein Literal abgebildet – alle anderen Variablen seien auf sich selbst abgebildet. Hierbei gilt anschaulich, dass eine Lösung für P genau dann eine Lösung von $\nu[P]$ darstellt, wenn beide Abbildungen denselben Bildwert für diese Variable annehmen.

Fasst man nun die jeweiligen Überlegungen zusammen, so gelangt man zu der Aussage, dass die Lösungen eines Basis-Graphmusters P, eingeschränkt auf die Variablen in $\nu[P]$, auch Lösungen für $\nu[P]$.

Lemma 4.5. *Seien $P \in \mathsf{BGP}$, ein RDF-Graph $G \in \mathcal{G}$ und eine Variablenabbildung ν gegeben. Die Aussage $\mu \in \Omega_P \Rightarrow \nu[\mu] \in \Omega_{\nu[P]}$ gilt, falls die folgenden Bedingungen erfüllt sind:*

i) $\forall x \in \mathrm{dom}(\mu) : \nu(x) = a, a \in \mathsf{RDF\text{-}T} \rightarrow \mu(x) = a$

ii) $\forall x, y \in \mathrm{dom}(\mu) : \nu(x) = \nu(y) \rightarrow \mu(x) = \mu(y)$

Beweis. Sei $\mu \in \Omega_P$ gegeben.

- Wenn die Variablenabbildung ν eine Variablenumbenennung durchführt, insbesondere gilt also $\nu : \mathbb{V} \rightarrow \mathbb{V}$ und $\forall x, y : x \neq y \rightarrow \nu(x) \neq \nu(y)$, dann ist offensichtlich $\nu[\mu] \in \Omega_{\nu[P]}$.
- Sei $x \in \mathrm{dom}(\mu)$ eine Variable mit $\nu(x) = a, a \in \mathsf{RDF\text{-}T}$, dann gilt nach Voraussetzung $\mu(x) = a$. Somit ist μ eine Lösung für ein Graphmuster, das durch Ersetzen von x durch a in P entsteht. Damit ergibt sich eine Situation wie zuvor.
- Seien $x, y \in \mathrm{dom}(\mu)$ zwei Variablen mit $\nu(x) = \nu(y)$, dann gilt nach Voraussetzung $\mu(x) = \mu(y)$. Somit ist μ eine Lösung für ein Graphmuster, das durch Ersetzen von y durch x in P entsteht. Damit ergibt sich eine Situation, die schon behandelt worden ist.

□

Betrachtet man die Umkehrung von Lemma 4.5, so stellt sich die Frage, ob aus der Kenntnis von ν und einer Lösungsabbildung von $\nu[P]$ auch eine für P konstruiert werden kann. Das Gegenbeispiel in Abbildung 4.8 widerlegt die Aussage für beliebige Definitionen von ν. Die Ursache hierfür findet sich in der Abbildung einer Variablen auf eine anonymen Ressource. Da in μ_2 keine Information über die Zuordnung der anonymen Ressource _:p zu einer Ressourcen oder einem Literal des RDF-Graphen enthalten ist, kann in μ_1 keine Belegung für ?person rekonstruiert werden. Nichtsdestotrotz kann gefolgert werden, dass es eine solche Belegung und damit ein μ_1 geben muss.

Selbst wenn der Wertebereich von ν keine anonymen Ressourcen enthält, kann des Weiteren festgestellt werden, dass nicht alle Lösungen für P aus denen von $\nu[P]$ abgeleitet werden können. Grund hierfür ist eine mögliche Abbildung zweier Variablen aus P auf dieselbe Variable oder denselben Ressource-IRI. Jedoch kann das folgende Lemma bewiesen werden:

G: { ex:einePerson foaf:currentProject ex:dbpedia . }

P_1: { ?person foaf:currentProject ?project . }

P_2: { _:p foaf:currentProject ?project . }

Abbildung 4.8: Obwohl $P_1 \sqsubset P_2$ gilt, kann die Lösung für P_1 nicht aus der von P_2 abgeleitet werden.

Lemma 4.6. *Seien $P \in \mathsf{BGP}$, ein RDF-Graph $G \in \mathcal{G}$ und eine Variablenabbildung ν gegeben. Es gelten die folgenden Aussagen:*

i) $\forall \mu \in \Omega_{\nu[P]} \exists \mu' \in \Omega_P : \nu[\mu'] \subseteq \mu$

ii) $\mathrm{ran}(\nu) \cap \mathbb{B} = \emptyset \Rightarrow \forall \mu \in \Omega_{\nu[P]} \exists \mu' \in \Omega_P : \nu[\mu'] = \mu$

Beweis. Zunächst wird ii) gezeigt. Sei also μ' wie folgt definiert:

$$\mu'(x) := \begin{cases} \mu(y) & \text{falls } \nu(x) = y, y \in \mathbb{V} \\ a & \text{falls } \nu(x) = a, a \in \mathbb{I} \cup \mathbb{L} \end{cases}$$

Um $\mu = \nu[\mu']$ zu zeigen, sind die Bedingungen aus Definition 4.3 zu überprüfen. Aufgrund der Definition von μ' ist sichergestellt, dass $\mu(y) = \mu'(x)$, falls $\nu(x) = y$ gilt. Diese stellt auch sicher, dass $\forall y, y' \in \mathrm{dom}(\nu) : \nu(y) = \nu(y') \Rightarrow \mu(y) = \mu(y')$ gilt.

Zuletzt ist $\mu' \in \Omega_P$ zu zeigen. Da μ eine Lösung für P ist, gibt es eine RDF-Instanzabbildung σ mit $\mu \circ \sigma[\nu[P]] \subset G$. Aus der Definition von μ' folgt $\mu'[P] = \mu[\nu[P]]$ und aus $\mathrm{ran}(\nu) \cap \mathbb{B} = \emptyset$ folgt schließlich $\mu' \circ \sigma[P] \subset G$ für dasselbe σ.

Wenn nun ν Variablen auf anonyme Ressourcen abbildet, folgt die Aussage i) aus obigen, da es nach Voraussetzung $\mu \in \Omega_{\nu[P]}$ eine RDF-Instanzabbildung σ geben muss, die eine solche anonyme Ressource auf einen Wert abbildet, so dass $\mu \circ \sigma[\nu[P]] \subset G$ gilt. Damit gibt es auch eine Lösung $\mu' \in \Omega_P$. □

Wie auch im vorangegangenen Abschnitt ergaben die Untersuchungen, dass eine Lösung eines Graphmusters P nur unter den genannten Bedingungen (Lemma 4.5) auch eine Lösung eines aus dem Anwenden einer Variablenabbildung entstandenen Graphmusters $\nu[P]$ sind. Wenn man hingegen die Lösungen von $\nu[P]$ betrachtet, so konnte festgestellt werden, dass man selbst im ungünstigsten Fall zumindest eine Teillösung von P erhält.

4.2.2.3 Lösungen von einem enthaltenden Graphmuster: $P_1 \sqsubset P_2$

Die Ausgangslage für diesen Abschnitt ist, dass die Lösungsabbildungen für ein Basis-Graphmuster P_1 über $G \in \mathcal{G}$ bekannt sind und dass dieses in einem anderen Graphmuster P_2 enthalten ist ($P_1 \sqsubset P_2$). Es wird im Folgenden der Fragestellung nachgegangen, wie Lösungen von P_1 und P_2 zueinander in Beziehung stehen.

Die Definition des Enthaltenseins eines Basis-Graphmusters basiert auf einer Graphmusterabbildung, die eine Hintereinanderausführung einer RDF-Instanzabbildung und einer Variablenabbildung darstellt (vgl. Definitionen 4.6

und 4.5). Aus den Lemmata 4.3 und 4.5 kann zunächst hergeleitet werden, dass diese beiden Abbildungen bestimmte Eigenschaften erfüllen müssen, damit eine Lösung des enthaltenen Graphmusters P_1 auch für P_2 Relevanz besitzt.

Selbst wenn die Bedingungen der Lemmata erfüllt sind, illustriert Abbildung 4.9, dass nicht jede Lösung von P_1 zu einer Lösung von P_2 erweitert werden kann. Das Graphmuster P_2 kann nämlich zusätzliche, die Lösungsmenge einschränkende Tripelmuster beinhalten.

```
G: { ex:einePerson foaf:currentProject ex:einProjekt .
     ex:einePerson foaf:knows ex:zweitePerson . }

P1: { ?person foaf:currentProject ?projekt . }

P2: { ?person foaf:currentProject ?projekt .
     ?person foaf:knows ex:einePerson . }
```

Abbildung 4.9: Obwohl $P_1 \sqsubset P_2$ kann nicht jede Lösung für P_1 kann zu einer Lösung von P_2 erweitert werden.

Wenn nun nicht jede Lösung von P_1 zu einer Teillösung von P_2 erweitert werden kann, stellt sich die Frage, ob nicht – wie man aus der Enthaltenseinsbeziehung folgern könnte – jede Lösung für P_2 auf einer Lösung von P_1 basiert. Das folgende Lemma zeigt genau dieses.

Lemma 4.7. *Seien $P_1, P_2 \in \mathsf{BGP}$ und ein RDF-Graph $G \in \mathcal{G}$ gegeben. Dann gilt das Folgende:*

$$P_1 \sqsubseteq P_2 \Rightarrow \forall \mu_2 \in \Omega_{P_2} \exists \mu_1 \in \Omega_{P_1} : \nu[\mu_1] \subseteq \mu_2$$

Beweis. Sei $\mu_2 \in \Omega_{P_2}$ beliebig gegeben. Wegen $P_1 \sqsubseteq P_2$ gibt es eine Graphmusterabbildung $\gamma = \nu \circ \sigma$ mit $\gamma[P_1] \subseteq P_2$. Daher muss μ_2 eine Lösung für $\gamma[P_1]$ über G beinhalten. Folglich muss es aufgrund der Lemmata 4.4 und 4.6 eine Lösung μ_1 für P_1 mit $\nu[\mu_1] \subseteq \mu_2$ geben. □

Unter welchen Bindungen aus einer gegebenen Lösungsabbildung μ_1 bzw. μ_2 die jeweils andere Lösungsabbildung konstruiert werden kann, wurde in den beiden voran gegangenen Abschnitten untersucht.

4.2.2.4 Lösungen von sich überlappenden Graphmuster: $P_1 \bullet\!\circ P_2$ und $P_1 \bullet\!\bullet P_2$

Die Ergebnisse des vorherigen Abschnitts lassen sich unmittelbar auf die Überlappenbeziehungen $\bullet\!\circ$ und $\bullet\!\bullet$ zwischen zwei Basis-Graphmustern P_1 und P_2 anwenden, da die Definitionen der Überlappenbeziehungen darauf basieren, dass beide Graphmuster ein gemeinsames enthalten. Aus diesem Grund muss beispielsweise eine Lösung $\mu_1 \in \Omega_{P_1}$, eingeschränkt auf das gemeinsame Graphmuster von P_1 und P_2, nicht zwangsläufig Teil einer Lösung $\mu_2 \in \Omega_{P_2}$ sein. Das Graphmuster P_2 könnte wiederum weitere einschränkende Graphmuster enthalten (vgl. Abbildung 4.9).

Darüber hinaus kann sogar ein einfaches Beispiel konstruiert werden (Abbildung 4.10), mit dem gezeigt werden kann, dass man aus der Kenntnis einer Lösung von P_1, eingeschränkt auf das gemeinsame Graphmuster, nicht einmal eine Teillösung des anderen Graphmusters P_2 gewonnen werden kann:

$\mu_1 = \{?person \to ex : einePerson, ?project \to ex : einProjekt\} \in \Omega_{P_1}$ und $\mu_2 = \{?person \to ex : zweitePerson, ?project \to ex : zweitesProjekt\} \in \Omega_{P_2}$

```
G: { ex:einePerson foaf:currentProject ex:einProjekt .
     ex:einePerson foaf:knows ex:zweitePerson .
     ex:zweitePerson foaf:currentProject ex:zweitesProjekt .
     ex:zweitePerson foaf:mbox <mailto:zp@example.de> . }

P1: { ?person foaf:currentProject ?projekt . }
     ?person foaf:mbox <mailto:zp@example.de> . }

P2: { ?person foaf:currentProject ?projekt .
     ?person foaf:knows ex:zweitePerson . }
```

Abbildung 4.10: Trotz beidseitigen Überlappens, $P_1 \bullet\!\!\bullet P_2$, gibt es keine Gemeinsamkeiten in den Lösungsabbildungen.

Die einzige Ausnahme bilden die eher pathologischen Fälle, in denen alle Lösungen des gemeinsamen Graphmusters P eine Teillösung von o.B.d.A. P_1 darstellen, z. B. wenn das Graphmuster P_1 ohne P nur Variablen und anonyme Ressourcen beinhaltet. In diesem Fall gibt es zu jeder Lösung von P_2 auch eine Lösung von P, die sowohl Teillösung von P_1 als auch P_2 ist.

Lemma 4.8. *Seien $P_1, P_2 \in \mathsf{BGP}$ und ein RDF-Graph $G \in \mathcal{G}$ gegeben. Wenn sich P_1 und P_2 einseitig mit einem $P : P \subseteq P_1 \wedge P \sqsubseteq P_2$ überlappen, dann gilt Folgendes:*

$$\forall \mu \in \Omega_P \exists \mu_1 \in \Omega_{P_1} : \mu \subseteq \mu_1 \Rightarrow \forall \mu_2 \in \Omega_{P_2} \exists \mu_1 \in \Omega_{P_1} \exists \mu \in \Omega_P : \mu \subseteq \mu_1 \wedge \mu \subseteq \mu_2$$

Beweis. Da nach Voraussetzung jede Lösung von P eine Teillösung von P_1 ist, folgt die Aussage unmittelbar aus Lemma 4.7. □

4.2.2.5 Lösungen der Vereinigung von Basis-Graphmustern: $P_1 \sqcup P_2$

Wie Abbildung 4.5 am Ende von Abschnitt 4.2.1 gezeigt hat, ist zwar eine mengenorientierte Vereinigung zweier Basis-Graphmuster möglich, aber wenn die Variablen der beiden Graphmuster in einem unbekannten, semantischen Zusammenhang stehen, ist sie nicht eindeutig. Daher liegt den Betrachtungen in diesem Abschnitt die Annahme zugrunde, dass eine konkrete Vereinigung $P_1 \sqcup P_2 = \gamma_1[P_1] \cup \gamma_2[P_2]$ mit gegebenen Graphmusterabbildungen γ_1 und γ_2 vorliegt.

Da wie im Lemma 4.2 gezeigt, die einzelnen Graphmuster in ihrer Vereinigung enthalten sind, liegt die Schlussfolgerung nahe, dass ihre jeweiligen Lösungen zu den Lösungen der Vereinigung beitragen.

Lemma 4.9. *Seien mit $P_1, P_2 \in \mathsf{BGP}$ zwei Graphmuster und ein RDF-Graph $G \in \mathcal{G}$ gegeben. Es gilt die folgende Aussage:*

$$\forall \mu \in \Omega_{P_1 \sqcup P_2} \exists \mu_1 \in \Omega_{P_1} \exists \mu_2 \in \Omega_{P_2} : \nu_1[\mu_1] \subseteq \mu \wedge \nu_2[\mu_2] \subseteq \mu$$

Beweis. Nach Definition der Vereinigung gibt es $\gamma_1 = \nu_1 \circ \sigma_1$ und $\gamma_2 = \nu_2 \circ \sigma_2$ mit $P_1 \sqcup P_2 = \gamma_1[P_1] \cup \gamma_2[P_2]$. Die Aussage folgt unmittelbar aus dem zweimaligen Anwenden des Lemmas 4.7. □

4.3 Indexauswahl

Die im vorherigen Abschnitt gewonnenen Erkenntnisse über die Beziehungen zwischen Basis-Graphmustern und deren Lösungen können nun für die Entscheidung eingesetzt werden, wann ein Index für die Berechnung einer Anfrage verwendet werden kann. Dazu wird Folgenden zunächst der Begriff eines *zulässigen Indexes* eingeführt. Im Anschluss daran wird der Fall betrachtet, dass für eine Anfrage mehrere zulässige Indexe existieren und diese gemeinsam zur Berechnung der Lösungen herangezogen werden.

4.3.1 Zulässige Indexe

Entsprechend der Definition 4.1 ist ein Index über ein Basis-Graphmuster definiert und materialisiert alle dessen Lösungen bezüglich eines RDF-Graphen. Intuitiv lässt sich ein Index für die Berechnung eines Basis-Graphmusters verwenden, wenn das Indexmuster ganz in dem Basis-Graphmuster der Anfrage enthalten ist.

Definition 4.12 (Zulässiger Index)**.** *Sei $G \in \mathcal{G}$ ein beliebiger RDF-Graph und $\mathcal{I} = I_1, \ldots, I_n$ die Menge der über G definierten Indexe. Des Weiteren sei Q eine Anfrage über G und $P_1, \ldots, P_m$ die Basis-Graphmuster von Q. Ein Index $I = (P_I, \Omega_{P_I}) \in \mathcal{I}$ ist* zulässig *bezüglich Q, falls $\exists P \in P_1, \ldots, P_m : P_I \sqsubseteq P$.*

Basierend auf den Einträgen des Indexes I lassen sich entsprechend des Lemma 4.7 die Lösungen der Anfrage Q berechnen. Wie in Abschnitt 4.2.2.4 gezeigt werden konnte ist eine teilweise Überlappung nur in pathologische Fälle geeignet.

Für die Untersuchung, ob ein Index zur Beantwortung einer Anfrage genutzt werden kann, sind dementsprechend die in der Anfrage enthaltenen Basis-Graphmuster von besonderem Interesse. Im Abschnitt 5.3 werden später Transformationsregeln beschrieben, um eine gute Voraussetzung zum Bestimmen der zulässigen Index zu schaffen, z. B. das Zusammenführen von Basis-Graphmuster.

Aus der Definition 4.6 (Enthaltensein) ergibt sich darüber hinaus, dass das Graphmuster eines zulässigen Indexes auch allgemeiner als das Basis-Graphmuster sein kann. Beispielsweise kann die Graphmusterabbildung eine Variable des Indexmusters auf eine Ressource des Anfragemusters abgebildet werden. Für die Anfrageausführung bedeutet dies, dass unter Umständen zusätzliche Filteroperatoren in den Ausführungsplan eingeführt werden müssen, um die nicht zur Lösung gehörenden Indexeinträge zu entfernen.

Beispiel 4.1. In Abbildung 4.11 zeigt die beiden Indexmuster P_{I_1} und P_{I_2} sowie das Anfragemuster P_Q. Während P_{I_1} mit der Graphmusterabbildung $\{?\texttt{project} \rightarrow \texttt{ex:dbpedia}\}$ zulässig bezüglich P_Q ist, kann es aufgrund des Objekts `"Schmidt"` für P_{I_2} keine Graphmusterabbildung γ mit $\gamma[P_{I_2}] \subseteq P_Q$ geben. □

4.3.2 Überdeckung

Wenn über einen RDF-Graph mehrere Indexe definiert sind, dann können für eine Anfrage mehrere zulässige Indexe existieren. Mit der folgenden Definition wird der Begriff der Überdeckung eingeführt:

P_{I_1} : { ?person foaf:currentProject ?project .
?person foaf:name ?name . }

P_{I_2} : { ?person foaf:currentProject ?project .
?person foaf:name "Schmidt" . }

P_Q : { ?person foaf:currentProject ex:dbpedia .
?person foaf:name ?name .
?person foaf:mbox ?mail . }

Abbildung 4.11: P_{I_1} ist zulässig bezüglich P_Q, P_{I_1} dagegen nicht

Definition 4.13 (Überdeckung eines Basis-Graphmusters)**.** *Seien ein Basis-Graphmustern P und eine Menge von zulässigen Indexen $\mathcal{I} = I_1, \ldots I_n$ bezüglich P gegeben. Falls es für eine Menge von Indexen $\mathcal{I}_C \subseteq \mathcal{I}$ eine Menge von Graphmusterabbildungen $\Gamma = \gamma_1, \ldots \gamma_k$ gibt, so dass $\bigcup_{i=1}^{k} \gamma_i[P_{I_c}] \sqsubseteq P$ mit $I_c \in \mathcal{I}_C$ gilt, dann ist das Paar $\mathcal{C} = (\mathcal{I}_C, \Gamma)$ eine* Überdeckung *von P bezüglich $\mathcal{I}$.*

In der Definition einer Überdeckung werden Graphmusterabbildungen verwendet, um eine Beziehung zwischen den Indexmustern und dem Basis-Graphmuster herzustellen. Dabei ist es auch erlaubt, dass für ein Indexmuster zwei verschiedene Graphmusterabbildungen existieren. Dieser Fall kann zum Beispiel auftreten, wenn ein Indexmuster mehrmals in dem Basis-Graphmuster vorkommt. Darüber hinaus kann es für dieselbe Menge an Indexen unterschiedliche Mengen von Graphmusterabbildungen eine Überdeckung bilden.

Beispiel 4.2. Abbildung 4.12 Das Indexmuster P_I des zulässigen Indexes I kommt in dem Graphmuster P zwei Mal vor. Die möglichen drei Überdeckungen sind:

- $\mathcal{C}_1 = (\mathrm{I}, \{?\mathtt{project} \rightarrow \mathtt{ex{:}dbpedia}\})$
- $\mathcal{C}_1 = (\mathrm{I}, \{?\mathtt{project} \rightarrow \mathtt{ex{:}onehunga}\})$
- $\mathcal{C}_1 = (\mathrm{I}, \{?\mathtt{project} \rightarrow \mathtt{ex{:}dbpedia}, ?\mathtt{project} \rightarrow \mathtt{ex{:}onehunga}\})$

□

P_I : { ?person foaf:currentProject ?project .
?person foaf:name ?name . }

P : { ?person foaf:currentProject ex:dbpedia .
?person foaf:currentProject ex:onehunga .
?person foaf:name ?name .
?person foaf:mbox ?mail . }

Abbildung 4.12: Drei mögliche Überdeckungen für das Graphmuster P und dem Index I

Weiterhin sei zu der Definition 4.13 noch angemerkt, dass sich die Graphmuster der Indexe einer Überdeckung nicht zwangsläufig überlappen müssen. Falls sich jedoch zwei Indexmuster überlappen, so müssen die Graphmusterabbildungen kompatibel sein. Das heißt, nach dem Anwenden der Graphmusterabbildung auf die Indexmuster muss die Vereinigung der entstehenden Graphmuster in dem Basis-Graphmuster enthalten sein. Das von der Vereinigung abgedeckte Graphmuster wird nachfolgend als *überdecktes Graphmuster* bezeichnet.

Nach der Definition der Überdeckung für ein einzelnes Basis-Graphmuster kann nun die Überdeckung für eine Anfrage definiert werden.

Definition 4.14 (Überdeckung einer Anfrage)**.** *Sei eine Anfrage Q mit den Basis-Graphmustern $P_1,\dots P_m$ und eine Menge von zulässigen Indexen $\mathcal{I} = I_1,\dots I_n$ gegeben. Falls $\mathcal{C}_1,\dots \mathcal{C}_m$ die Überdeckungen der Graphmuster $P_1,\dots P_m$ sind, so ist die Menge $\mathcal{C}_Q = \{\mathcal{C}_1,\dots \mathcal{C}_m\}$ eine Überdeckung der Anfrage Q.*

4.4 Index-Management

Um sich dem Abschätzen der Kosten für einen Indexzugriff anzunähern, wird zunächst das Speichermodell für das Verwalten der Indexeinträge beschrieben. Im Resource Centered Store stellt ein Index die Materialisierung der Lösungsabbildungen eines Basis-Graphmusters bezüglich eines RDF-Graphen dar. Das Verwenden eines Indexes entspricht dabei dem Lesen aller gespeicherten Lösungsabbildungen. Da eine Lösungsabbildung aus einer Menge von Variablenabbildungen – der Zuordnung eines Wertes zu einer Variablen – besteht, kann die Menge von Lösungsabbildungen für ein Indexmuster gut als Tabelle repräsentiert werden. Jede Spalte beinhaltet die Variablenbindungen für eine Variable, jede Zeile entspricht einer Lösungsabbildung.

Beispiel 4.3. In Abbildung 4.13 werden für einen RDF-Graph und einen Index die zugehörigen Lösungsabbildungen in tabellarischer Form repräsentiert. □

RDF-Daten

```
ex:Schmidt foaf:currentProject ex:dbpedia .
ex:Schmidt foaf:currentProject ex:onehunga .
```

Indexmuster

```
{ ?person foaf:currentProject ?project . }
```

Lösungsabbildungen

?person	?project
ex:Schmidt	ex:dbpedia
ex:Schmidt	ex:onehunga

Abbildung 4.13: Tabellarische Darstellung von Lösungsabbildungen

4.4.1 Speichermodell

Auf Grund der Ähnlichkeit zum relationalen Datenmodell und den nachfolgend aufgelisteten Vorteilen werden Indexe im RCS in einem relationalem DBMS, PostgreSQL [Pos13], verwaltet.

(i) Die Technologien von RDBMS sind bezüglich der Verwaltung und Auslesen der Daten sowie Caching-Mechanismen ausgereift.

(ii) Falls eine SPARQL-Anfrage einen Spezialfall eines Indexmusters darstellt, können die Lösungen des Indexes durch zusätzliche SQL-Prädikate nachgefiltert werden, z. B. Binden einer Variable des Indexmusters an einen RDF-Term der Anfrage.

(iii) Falls eine Überdeckung mehrere Indexe beinhaltet, deren Indexmuster sich überlappen, dann können deren Einträge innerhalb des RDBMS verknüpft werden.

(iv) Da die Lösungen in Tabellen gespeichert werden, können über deren Spalten Indexe angelegt werden, die die Performanz von Filter- und Join-Operationen über verbessern können, z. B. Verwenden von mehreren Indexen zur Anfragebearbeitung.

Wie bereits durch Abbildung 4.13 veranschaulicht, werden die Indexeinträge in deiner Tabelle gespeichert, deren Spalten den Variablen des Indexmusters entsprechen. Die Einträge in der Tabelle sind dabei normalisierte Werte, d.h., anstelle von IRIs und Literalen werden IDs gespeichert. Dies reduziert zum einen den benötigten Speicherplatzbedarf und zum anderen können die Wert unmittelbar vom RCS weiterverarbeitet werden. Neben diesen Tabellen wird außerdem eine Tabelle für das Verwalten der Abbildung zwischen dem Indexmuster und der Tabelle mit dessen Lösungsabbildungen benötigt. In dieser Tabelle werden die folgenden Daten gespeichert:

- `uri` – eindeutiger Identifikator eines Indexes
- `pattern` – zugehörendes Indexmuster
- `table_name` – Name der Tabelle mit den Indexeinträgen
- `graph_uris` – IRIs der dem Index zugrunde liegenden Graphen

4.4.2 Verwenden eines Indexes

Im Folgenden wird davon ausgegangen, dass bereits ein Ausführungsplan zur Beantwortung einer SPARQL-Anfrage vom Anfrageoptimierer erstellt worden ist,[1] der ein oder mehrere Indexoperatoren beinhaltet. Da die Indexeinträge in einem RDBMS verwaltet werden, müssen entsprechende SQL-Anfragen generiert werden. Dabei werden zum Lesen der Indexeinträge die Aspekte (i) des Einschränkens auf die benötigten Variablen, (ii) des Nachfilterns der Indexeinträge und (iii) des Überlappens von Indexmustern berücksichtigt.

Im einfachsten Fall beinhaltet der Ausführungsplan den Zugriff auf einen Index, dessen Muster sich mit keinem anderen Indexmuster überlappt und

[1] Die Generierung von Ausführungsplänen wird später in Abschnitt 5.5 erläutert.

dessen Lösungsabbildungen ohne jegliche Einschränkungen an die konsumierenden Operatoren weitergegeben werden. In diesem Fall wird die folgende SQL-Anfrage generiert: `SELECT * FROM <table_name>`, wobei `table_name` dem Namen der zum Index gehörenden Tabelle entspricht.

Falls der Anfrageoptimierer feststellt, dass nicht alle Variablenbindungen von den konsumierenden Operatoren benötigt werden (vgl. Abschnitt 5.3.3), dann kann die Menge der zu transferierenden Daten durch eine entsprechende SQL-Anweisung eingeschränkt werden: `SELECT <vars> FROM <table_name>`. Dabei entspricht `vars` der Liste der Variablennamen, deren Bindungen für die nachfolgenden Operatoren erforderlich sind.

Ein Nachfiltern der Indexeinträge ist erforderlich, falls das Indexmuster allgemeiner als das überdeckte Graphmuster der Anfrage. Das heißt, die zugehörende Graphmusterabbildung bildet eine Variable des Indexmusters auf eine Konstante des Anfragemusters ab (vgl. Abschnitt 4.3.1). Für den Fall, dass alle Variablenbindungen zurückgegeben werden, ergibt sich die folgende SQL-Anfrage:

```
SELECT * FROM <table_name>
WHERE v1 = <c1> AND ... AND vn = <cn>
```

In dieser Anfrage sind `v*` und `c*` die Variablen des Indexmustern bzw. die zugehörigen Konstanten des Anfragemusters. In besonderen Situationen können darüber hinaus auch komplexere Filterprädikate entstehen, die `OR`-Operator beinhalten (vgl. nachfolgendes Beispiel).

Beispiel 4.4. Der in Abbildung 4.14 dargestellte Index kann zur Beantwortung der gezeigten SPARQL-Anfrage[2] verwendet werden. Um einen unnötigen Transfer von Daten zu vermeiden, wird der Filterausdruck im relationalen DBMS ausgewertet (s. SQL-Anfrage in Abb. 4.14). □

Indexmuster

```
{ ?person foaf:currentProject ?project . }
```

SPARQL-Anfrage

```
SELECT * WHERE {
      ?person foaf:currentProject ?p .
      FILTER (?p = ex:dbpedia || ?p = ex:onehunga) .
}
```

SQL-Anfrage

```
SELECT * FROM <table_name> WHERE {
      project = 'ex:dbpedia' OR project = 'ex:onehunga') .
}
```

Abbildung 4.14: SQL-Anfrage zum Indexzugriff enthält `OR`-Operator

Der letzte Aspekt bei der Generierung der SQL-Anfrage betrifft Indexe mit sich überlappenden Graphmustern. In dieser Situation kann die Anfragekom-

[2] Zum besseren Verständnis zeigt die Anfrage IRIs anstelle der normalisierten Werte.

ponente des relationalen DBMS eingesetzt werden, um die Einträge dieser Indexe miteinander zu verknüpfen. Zum einen wird dadurch die Menge der zum RCS zu transferierenden Daten reduziert und zum anderen können die inhärenten Join-Algorithmen und Statistiken ausgenutzt werden. Zum Generieren der SQL-Anfrage werden zunächst auf Basis der Graphmusterabbildungen die Join-Prädikate bestimmt. Die auf denselben RDF-Term abgebildeten Variablen der Indexmuster werden hierzu gleichgesetzt. Für zwei sich überlappende Indexe hat die Anfrage das folgende Aussehen:

```
SELECT * FROM <table_name_r> r, <table_name_s> s
WHERE r.v1 = s.w1 AND ... AND r.vn = s.wn
```

Bei `v*` und `w*` handelt es sich um Variablen der jeweiligen Indexe.

Das Verwenden von Indexen bei der Beantwortung einer Anfrage kann alle zuvor beschriebenen Aspekte involvieren. Das Projizieren auf Variablen, das Nachfiltern der Indexeinträge sowie das Verknüpfen von Lösungsabbildungen verschiedener Indexe sind bezüglich der Generierung orthogonal zueinander und können miteinander kombiniert werden.

4.4.3 Aktualisieren von Indexen

Nachdem im RDF-Graph Tripel hinzugefügt oder gelöscht wurden, müssen alle über dem betroffenen RDF-Graph definierten Indexe auf Aktualität hin überprüft werden. In diesem Abschnitt wird die Frage untersucht, wie mit möglichst geringem Aufwand die Indexe aktuell gehalten werden können. Zunächst wird das Einfügen eines Tripels betrachtet, anschließend wird auf das Löschen eingegangen.

Das Einfügen eines Tripels in einen RDF-Graph kann zur Folge haben, dass zusätzliche Lösungsabbildungen für einige Indexe entstehen. Die bereits im Index existierenden Lösungsabbildungen sind von dieser Operation nicht betroffen, da es sich bei den Indexmustern um Basis-Graphmuster handelt. Der Grund dafür ist, dass die Existenz eines Tripels nicht die ein Graphmuster erfüllenden Teilgraphen in dem RDF-Graph verändert.

Sei also eine Menge $\mathcal{I} = \{I_1, \ldots, I_n\}$ über einen RDF-Graph G und ein neu eingefügtes Tripel t gegeben. Das Aktualisieren der Indexe erfolgt in zwei Schritten: Zuerst werden die zu aktualisierenden Indexe ermittelt und anschließend die hinzuzufügenden Lösungen berechnet. In dem ersten Schritt wird jedes Indexmuster dahingehend überprüft, ob das Tripel t Teil einer Lösung für dieses Muster sein kann. Für einen Index $I = (P, \Omega_P)$ ist nur dann der Fall, falls es eine Lösungsabbildung μ und eine RDF-Instanzabbildung σ mit $\mu \circ \sigma[t_I] = t$ mit $t_I \in P$ gibt.

Zum Berechnen der zum Index hinzuzufügenden Lösungen (im zweiten Schritt) müssen diejenigen Teilgraphen des RDF-Graphen bestimmt werden, die das Tripel t beinhalten und dem Indexmuster P genügen[3]. Dazu werden als erstes alle möglichen Lösungen $\mu_1, \ldots \mu_k$ für alle $t_P \in P$ und t bestimmt. Anschließend werden für die Lösungen für die Muster $\mu_1(P), \ldots \mu_k(P)$ berechnet. Zu jeder Lösungen werden dann die Variablenbindungen aus der angewendeten Lösungsabbildung hinzugenommen, um eine Lösung für das Indexmuster P zu erhalten. Diese wird zum Schluss zum Index hinzugenommen.

[3] Anmerkung: Da jedes Tripel nur einmal in einem RDF-Graph vorkommen darf, kann kein Index eine Lösung enthalten, die aufgrund des Tripels t entstanden ist

Wenn ein Tripel t aus einem RDF-Graph gelöscht wird, wird genau wie beim Einfügen vorgegangen, nur dass am Ende die ermittelten Lösungen aus dem Index gelöscht werden. Das heißt, es werden die Lösungen ermittelt, die hinzugenommen werden würden, wenn das Tripel t zum Graph hinzugefügt worden wäre. Diese werden dann aus dem Index entfernt. Ein direktes Löschen aus dem Index ist nicht möglich, da im Allgemeinen nicht bestimmt werden kann, aufgrund welcher Tripel eine Lösung in den Index aufgenommen worden war. Das folgende Beispiel zeigt einen solchen Fall:

Beispiel 4.5. Abbildung 4.15 zeigt einen Index und einen RDF-Graph. Falls im RDF-Graph das erste Tripel gelöscht werden würde, kann auf Basis von Graphmuster und gelöschtem Tripel allein nicht entschieden werden, welche Lösungsabbildungen aus dem Index gelöscht werden können. Dahingegen kann anhand der Lösungsabbildung und des zu löschenden Tripels die Menge der einzufügenden Lösungen bestimmt werden: μ :?*person* → `ex:Schmidt`, wobei die zugehörige RDF-Instanzabbildung die anonyme Ressource auf `ex:dbpedia` abbildet. □

Index

```
{ ?person foaf:currentProject [] . }
```

?person
ex:Schmidt
ex:Schmidt

RDF-Graph

```
{ ex:Schmidt foaf:currentProject ex:dbpedia .
  ex:Schmidt foaf:currentProject ex:onehunga . }
```

Abbildung 4.15: Löschen von Lösungen nicht auf Basis von Indexmuster und Tripel möglich

4.5 Kostenabschätzung

Über den Begriff der Überdeckung werden diejenigen Indexe definiert, die für die Bearbeitung einer Anfrage potentiell verwendbar sind. In diesem Abschnitt wird diskutiert, ob aus Sicht der Ausführungskosten der Einsatz eines Indexes auch sinnvoll ist. Auf der einen Seite reduziert das Verwenden eines Indexes über einem größeren Graphmuster die Anzahl der erforderlichen Joins. Auf der anderen Seite können Join-Operationen zwischen den Indexen einer Überdeckung oder ein Nachfiltern der Indexeinträge erforderlich sein. Die Join-Operationen und das Nachfiltern wirken dem Nutzen der Indexe entgegen.

Damit der Anfrageoptimierer über die Verwendung einer Überdeckung entscheiden kann, werden im Folgenden die Ausführungskosten von Überdeckungen betrachtet. Ein wesentliches Maß hierfür ist die Anzahl der zuzugreifenden Datenseiten, die für das Lesen der Indexeinträge erforderlich sind. Dieser Wert hängt im Wesentlichen von dem zugrunde liegenden relationalem DBMS ab. Ein weiteres Maß stellt die Kardinalität der an den nachfolgenden Operator weitergegebenen Lösungsabbildungen dar. Dieser Wert wird durch ein eventuell notwendiges Nachfiltern der Indexeinträge bestimmt.

Beim Bestimmen der Kosten können zwei Wege eingeschlagen werden: Entweder es wird der Anfrageoptimierer des relationalen DBMS (mittels des `EXPLAIN`-Schlüsselwortes) oder des RCS verwendet. Während ersterer Informationen über die Konfiguration des Datenbanksystems und der verwendeten Algorithmen in die Kalkulation einbeziehen kann, verfügt letzterer über statistische Informationen über die RDF-Daten (vgl. Abschnitt 5.6).

In der vorliegenden Konfiguration erscheint die Verwendung des Anfrageoptimierer der relationalen Datenbank geeigneter. Im Falle des Zugriff auf einen einzelnen Index liegen dem Anfrageoptimierer des RCS hinreichend viele Informationen vor, um die Kosten abzuschätzen (s. nächsten Abschnitt 4.5.1). Bereits wenn zwei überlappende Indexe zugegriffen werden, bietet der Anfrageoptimierer des RDBMS eine präzisere Abschätzung. Auf der einen Seite können die Kosten für die Join-Operationen besser abgeschätzt werden, da der verwendete Algorithmus bekannt ist. Auf der anderen Seite können für die Kardinalität der Join-Operation datenbankinterne Statistiken herangezogen werden. Bei einer Überdeckung mit mehr als zwei sich überlappenden Indexen kommt zudem der Aspekt der Wahl der Join-Reihenfolge hinzu.

Im den beiden folgenden Abschnitten wird dennoch diskutiert, wie mit Hilfe der dem RCS zur Verfügung stehenden statistischen Informationen die Kosten bewertet werden kann. Zunächst wird der Fall eines einzelnen Indexes betrachtet, anschließend werden Indexe mit sich überlappenden Graphmustern untersucht.

4.5.1 Einzelner Index

Die wesentlichen Kosten für den Zugriff auf einen einzelnen Index entstehen durch das Laden der Datenseiten des Indexes vom Sekundärspeicher. Falls das Indexmuster nicht allgemeiner als das überdeckte Graphmuster ist – die Graphmusterabbildung bildet Variablen auf Variablen ab –, dann müssen immer alle Indexeinträge gelesen werden. Dies entspricht einem Table-Scan-Operator in dem relationalem DBMS und demzufolge ist für die Anzahl der allozierten Datenseiten ausschlaggebend.

Die Anzahl der benötigten Datenseiten wird zum einen durch die Anzahl der im Indexmuster enthaltenen Variablen und der Anzahl der Lösungsabbildungen bestimmt. Da in einer Indextabelle nur normalisierte Werte gespeichert werden und im Allgemeinen die Anzahl der Indexeinträge deutlich größer als die Anzahl der Variablen ist, wächst der Speicherplatzbedarf linear mit der Indexgröße, d.h. $O(|I|)$. Des Weiteren werden die Indexeinträge kompakt auf den Datenseiten (Ausnutzung ca. 100 %) gespeichert. Aus diesen Gründen eignet sich die Anzahl der Indexeinträge zum Abschätzen der Kosten. Um die Kosten eines Indexzugriffs mit denen einer Auswertung ohne Indexe (vgl. Abschnitt 5.6) vergleichen zu können, wird die Anzahl der Datenseiten eines Indexes $I = (P, \Omega_P)$ wie folgt abgeschätzt:

$$|\mathrm{p}_I| = \frac{k \cdot |\Omega_P| \cdot \mathrm{var}(P)}{||page||}$$

In der Formel wird der Overhead für das Verwalten von Metadaten pro Datenseite vernachlässigt und eine ID-Größe von k Bytes angenommen.

Die Kardinalität wird von den Filterprädikaten zum Nachfiltern der Indexeinträge bestimmt. Falls kein Nachfiltern erforderlich ist, entspricht die

Kardinalität offensichtlich der Anzahl an Indexeinträgen. Im Falle eines Nachfilterns werden statistische Informationen herangezogen, um die Selektivität der Filterprädikaten abzuschätzen. Wie später in Abschnitt 5.6 erläutert wird, stehen Histogramme für Tripelmuster zur Verfügung, die die Kombinationen SP, OS und PO abdecken. Da diese Histogramme für die Daten des zugrunde liegenden RDF-Graphen erhoben worden sind, wird im Folgenden angenommen, dass sich die Werte im Index ähnlich zu denen im RDF-Graph verhalten. Unter Annahme der Unabhängigkeit der Filterprädikate ergibt sich somit die folgende Abschätzung für einen Index I und einer Graphmusterabbildung γ:

$$\text{card}(I) = |I| \cdot \text{sel}(\gamma(P_I))$$

Durch das Anwenden der Graphmusterabbildung γ wird das Indexmuster so transformiert, dass es dem überdeckten Graphmuster in der Anfrage entspricht. Für dieses kann mit dem in Abschnitt 5.6 beschriebenen Verfahren die Selektivität das Graphmusters bestimmt werden.

4.5.2 Kosten einer Überdeckung

Beim Betrachten der Kosten einer Überdeckung genügt es, die Indexe mit überlappenden Graphmustern zu betrachten. Falls sich die Indexmuster nicht überlappen, wird ein Kreuzprodukt über die Lösungsabbildungen berechnet. Im Falle von sich überlappenden Indexmustern müssen dahingegen Join-Operationen ausgeführt werden, wobei die Einflussfaktoren für die Kosten dieselben sind, wie in einem relationalem DBMS, z. B. Sekundärspeicherzugriff, Kardinalität der Relation sowie Join-Reihenfolge. Auf der Basis der im vorherigen Abschnitt dargestellten Kosten für den Zugriff auf einen einzelnen Index können die für relationale DBMS etablierten Verfahren und Algorithmen (z. B. [SAC+79, Ioa96, Cha98]) eingesetzt werden, um die Kosten für eine Überdeckung abzuschätzen. Hierbei können die Histogramme über die Join-Paare von Tripelmustern miteinbezogen werden, die in Abschnitt 5.6 beschrieben werden.

4.5.3 Heuristiken für die Wahl von Indexmustern

Die Wahl des Indexmusters bestimmt, inwiefern ein Index für die Anfragebearbeitung sinnvoll eingesetzt werden kann. Der grundlegende Gedanke bei der Erstellung eines Indexes ist häufig wiederkehrende und mittels vieler Join-Operationen aufwändig zu berechnende Graphmuster von Anfragen vorzuberechnen. In diesem Abschnitt werden Heuristiken für die Wahl eines geeigneten Indexmusters beschrieben.

Keine sternförmigen Muster. Das Indexmuster sollte nicht hauptsächlich aus einem sternförmigen Graphmuster bestehen. Die Motivation für diese Heuristik liegt im Speichermodell des Resource Centered Stores: Für die Berechnung der Lösungsabbildungen für ein Sternmuster werden keine Join-Operationen benötigt, da alle dafür benötigten Tripel auf einer einzigen Datenseite gehalten werden. Darüber hinaus können die zuzugreifenden Datenseiten über das Kombinieren von Prädikat- und Objektindexen eingeschränkt werden.

Ein Indexieren eines Sternmusters wäre nur in dem eher pathologischen Fall sinnvoll, wenn im RCS die zu den Lösungen einer Anfrage beitragenden Subjekte gleichmäßig über alle Datenseiten verteilt sind. In diesem Fall müssten ohne Index alle Datenseiten gelesen und ausgewertet werden. Ähnlich dazu wäre theoretisch eine Situation denkbar, in der das Kombinieren von Prädikat- und Objektindex die Menge der Datenseiten nur unwesentlich einschränkt.

Beispiel 4.6. Wenn in einem RDF-Graph beispielsweise nach Lösungsabbildungen für das Tripelmuster {?s p o} gesucht wird, dieser jedoch nur wenige passende Tripel und viele Teilgraphen der Form {?s p ?o; ?p o} enthält, dann wäre die Definition eines Indexes über Graphmuster sinnvoll, das das Tripelmuster {?s p o} beinhaltet. □

Kostenreduktion. Die Definition eines Indexes ist für solche Graphmuster zweckmäßig, denen nur wenige Teilgraphen im RDF-Graph entsprechen, aber deren Bestimmung kostenintensiv ist. Hohe Kosten können zum einen entstehen, wenn durch das Kombinieren von Prädikat- und Objektindex die zuzugreifen Datenseiten nicht gut eingeschränkt werden können (vgl. Beispiel 4.6. Zum anderen kann das Verknüpfen von Tripelmustern (Join-Operation) viele Datenseiten zugegriffen bzw. große Zwischenergebnisse erfordern.

Beispiel 4.7. Abbildung 4.16 zeigt eine Anfrage mit dem Prädikat `rdf:type`, das typischerweise alle Ressourcen in einem RDF-Graph besitzen. Um die Lösungen der einzelnen Tripelmuster miteinander Verknüpfen zu können, müssten daher wahrscheinlich alle Datenseiten gelesen werden. □

```
SELECT ?person WHERE {
  ?other rdf:type ?type
  ?person rdf:type ?type .
  ?person foaf:name "Schmidt" .
}
```

Abbildung 4.16: Kostenintensive Anfrage aufgrund des Prädikats `rdf:type`

Häufige Graphmuster. In Ergänzung zur Heuristik der Kostenreduktion sind insbesondere häufig in Anfragen auftretende Graphmuster von Interesse, da dadurch ein Index häufiger vom Anfrageoptimierer in Betracht gezogen bzw. in Ausführungsplänen verwendet wird. Damit ergibt sich ein höherer Nutzen für den Aufwand der Aktualisierung und dem für den Index erforderlichen Speicherplatz. In Anfragen selten auftretende Indexmuster führen dahingegen zu spezialisierten Indexen. Idealerweise sollten bei der Wahl der Indexmuster eine Kombination dieser und der vorherigen Heuristik angestrebt werden, um häufig angefragte und selektive Indexmuster zu erhalten.

4.6 Verwandte Arbeiten

Die überwiegende Teil der existierenden RDF-Managementsysteme legt bei der Verwaltung von RDF-Daten den Fokus auf das Indexieren von Tripeln bzw. Quadrupel. Nur wenige Ansätze verwenden Pfade oder häufige Graphmuster als Basis für das Indexieren, wie sie beispielsweise für objektorientierten, Graph- und XML-Datenbanken entwickelt worden sind. In objektorientierten Datenbanken wurden diese dazu genutzt, um beim Evaluieren von Pfadausdrücken über Eigenschaften von Entitäten eine ähnliche Performanz wie relationale Datenbanksystem zu erreichen. Graphdatenbanken zielen auf die Verwaltung von Objekten mit beliebig vielen Eigenschaften ab, ohne dass vorher das Schema der Objekte definiert werden muss. Viele Indexierungsverfahren in diesem Bereich arbeiten auf einer größeren Menge von kleineren, nicht miteinander verknüpften Graphen wie beispielsweise Molekülen und liefern diejenigen Graphen, die (potentiell) einen gesuchten Teilgraphen enthalten könnten. Auf XML-Dokumente spezialisierte Datenbankmanagementsysteme nutzen für das Indexieren die Baumstruktur der Daten aus. Der Fokus hierbei liegt vor allem auch auf dem effizienten Bestimmen von Beziehungen zwischen Elementen (z. B. Reihenfolge, Vorgänger- und Nachfolgerbeziehungen).

Obwohl mittels RDF-Schema das verwendete Vokabular bekannt ist und auch Definitions- und Wertebereiche von Eigenschaften ermittelt werden können, kann sich ein Indexierungsverfahren für RDF nicht darauf verlassen. Da die Vokabulare in der Offenheit des Internets definiert sind, können jederzeit dem RDF-Managementsystem zuvor unbekannte Konzepte und Eigenschaften auftreten. Darüber hinaus erlaubt RDFS nicht einmal eine Festlegung Kardinalität einer Eigenschaft. Aus diesen Gründen werden in diesem Abschnitt Indexierungsverfahren betrachtet, die ohne Schemakenntnisse arbeiten.

Im Folgenden werden die Indexierungsverfahren nicht anhand des Datenmodells unterschieden, für die sie entwickelt worden sind, sondern nach ihrer Funktionsweise. Dies erscheint sinnvoll, da alle betrachteten Datenmodelle grundsätzlich auf Graphen basieren. Es werden dementsprechend im Folgenden zunächst Verfahren betrachtet, die auf dem Indexieren von Pfaden basieren. Der daran anschließenden Abschnitt befasst sich mit Verfahren, die in Graphen enthaltene Strukturen geeignet zusammenfassen, so dass diese für die Begrenzung des Suchraumes beim Evaluieren eine Anfrage eingesetzt werden können.

4.6.1 Materialisierung von Graphmustern

In diesem Abschnitt werden Ansätze vorgestellt, die nicht das Indexieren aller Tripel eines RDF-Graphen zum Ziel haben, sondern nur die Lösungsabbildung von ausgewählten Graphmustern materialisieren und zur Beantwortung von Anfragen verwenden.

RDFMatView. Aufbauend auf den Arbeiten von Heese [HLQR07] und Rothe [Rot12] wurde von Espinola ein Ansatz entworfen, wie SPARQL-Anfragen mittels Materialisierung der Lösungsabbildungen von Graphmustern beantwortet werden können. Das in [Esp12] entwickelte System basiert auf dem Jena2-Framework, wobei sowohl die RDF-Daten als auch die materialisierten

Lösungsabbildungen in einer relationalen Datenbank gespeichert werden. Espinola untersuchte in seiner Arbeit drei Verfahren: (i) Verknüpfen von Indexen außerhalb des RDBMS mittels des Jena Frameworks (ii) Übersetzen der SPARQL-Anfrage in SQL und Ausführen im RDBMS und (iii) Hybride Anfrageausführung der beiden vorherigen Ansätze

Im Vergleich zu dem in dieser Arbeit vorgestellten Ansatz unterscheidet sich die Arbeit von Espinola in den folgenden Punkten:

- *Datenhaltung.* In [Esp12] werden die RDF-Daten in einem relationalen DBMS gehalten. Ein Bezug zwischen materialisierten Graphmustern und einem nativen Speichermodell wurde nicht hergestellt.

- *Normalisierung.* In Espinolas Arbeit werden die Werte der Variablenabbildungen der materialisierten Graphmuster nicht normalisiert.

- *Kostenmodell.* Das Kostenmodell von Espinola basiert auf der Größe des Index, der Häufigkeit des materialisierten Graphmusters und der Größe des RDF-Graphen. In dieser Arbeit hingegen werden Histogramme zum Abschätzen der Kardinalitäten von Join-Operationen herangezogen. Diese werden auch genutzt, um die gleichzeitige Auswertung von Index und Filterausdruck im relationalen DBMS zu bewerten.

- *Anfrageoptimierung.* Im Gegensatz zu [Esp12] wird in dieser Arbeit die ganzheitliche Optimierung von Anfragen und Indexzugriff beleuchtet.

- *Filterausdrücke.* In der Arbeit von Espinola wird die Kombination von Indexzugriff und Filterausdrücken nicht diskutiert.

- *Heuristiken.* In dieser Arbeit werden Heuristiken für die Wahl von günstigen Indexmustern beschrieben.

4.6.2 Tripel-/Quadrupelbasiert

In diesem Abschnitt werden Ansätze vorgestellt, die auf dem Indexieren der in einem RDF-Graphen enthaltenen Tripel basieren. Dabei werden unterschiedliche Kombinationen von Subjekt, Prädikat, Objekt und gegebenenfalls Kontext indexiert. Als vereinfachte Notation für die einzelnen Komponenten eines Tripels werden im Folgenden die Buchstaben S, P, O und C verwendet.

Kombinierte Schlüssel in B-Bäumen. Harth und Decker verwenden sechs B-Bäume um benannte RDF-Graphen zu indexieren, da diese unter anderem auch Bereichs- und Präfixanfragen unterstützen. [HD05] Die Einträge in den Indexen bestehen aus verschiedenen Kombinationen von Subjekt, Prädikat, Objekt und Graph-URI, genau genommen SPOC, POC, OCS, CSP und OS. Damit lassen sich unabhängig von den Variablenpositionen alle möglichen Tripelmuster mit Hilfe von einem dieser Indexe beantworten. Darüber hinaus benutzen sie spezielle Quadrupel, um Statistiken über die Anzahl der Vorkommen für bestimmte Tripelmuster vorzuhalten.

Sechstupel Indexierung. Ähnlich zu dem vorherigen Ansatz wird in [WKB08] ein Ansatz beschrieben, in dem sechs Kombinationen von Subjekt, Prädikat und Objekt indexiert werden (SPO, SOP, PSO, POS, OSP und OPS). Im Unterschied zu Harth und Decker werden in diesem Ansatz Verknüpfungen mit den in Beziehung stehenden Subjekten, Prädikaten bzw. Objekten ebenfalls im Index abgelegt. Beispielsweise zeigt Abbildung 4.17 den Indexeintrag für das Subjekt s_i, das mit den in Beziehung stehenden Prädikaten und Objekten verknüpft ist. Die Autoren bezeichnen diese Systeme als Hexastore.

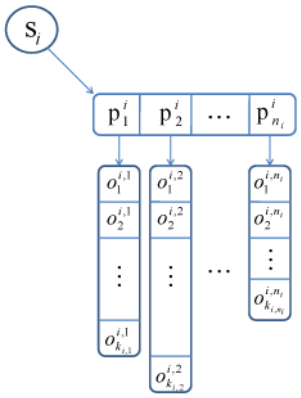


Abbildung 4.17: SPO-Index: Prädikate und Objekte sind mit dem Subjekt verknüpft [WKB08]

Obwohl dieser Ansatz den Charakteristiken von RDF-Daten (mehrwertige oder optionale Prädikate) Rechnung trägt, besteht ein wesentlicher Nachteil dieses Ansatzes im redundanten Speichern von Informationen. Auch wenn Listen von Subjekte, Prädikate bzw. Objekte zwischen Indexen geteilt werden können, wird in [WKB08] ein fünffach höherer Speicherbedarf gegenüber einer Tripeltabelle festgestellt.

Bitmat. In diesem Ansatz [ACZH11] werden die RDF-Tripel als ein dreidimensionaler Bit-Raum interpretiert, wobei die Subjekte, Prädikate und Objekte jeweils eine Dimension formen. Dabei bedeutet ein gesetztes Bit in diesem Raum die Existenz eines Tripel im RDF-Graphen mit dem entsprechenden Subjekt, Prädikat und Objekt. Aus der dreidimensionalen Struktur werden in dem Ansatz zweidimensionale Strukturen (Bitmat) erzeugt und gespeichert. Dazu wird zunächst für jedes Prädikat die SO-Matrix extrahiert und anschließend die Transponierte (OS-Matrix) generiert. Für die Subjekt- und Objekt-Dimension werden nur die Matrizen PO und PS erstellt. Damit erhält man Indexe für die Kombinationen PSO, POS, SPO und OPS. Aufgrund der Charakteristiken von RDF-Daten sind die generierten Bitmats schwach besetzt und können per *run length encoding* gut komprimiert werden. Darüber hinaus werden die Ressourcen und Literale auf einen ganzzahligen Identifikator abgebildet, um die Evaluation von Anfragen zu beschleunigen.

Dieser Indexierungsansatz weist zwei Einschränkungen auf: Zum einen werden Join-Operationen über SP und PO nicht unterstützt, da hierfür keine Bitmats erstellt werden. Zum anderen ist das Hinzufügen von Tripeln mit einer neuen Ressource kostenintensiv, da in einem solchen Fall ein Neuberechnen aller Bitmats erforderlich ist. Atre et al. definieren über die Bitmats eine Menge von Operatoren, die das Beantworten von Anfragen auf Basis der Bitmats ermöglichen.

GRIN. In [UPS07] beschreiben die Autoren einen Ansatz, bei dem zunächst der RDF-Graph in eine Menge von disjunkten Cluster von Ressourcen aufgeteilt wird. Die Cluster werden anhand einer zentralen Ressource und den

Abständen zu den anderen Ressourcen bestimmt – hierzu wird der PAM-Algorithmus (Partitioning Around Medoids) verwendet. Die Cluster werden anschließend in eine einen balancierten Binärbaum überführt, der zum Auffinden der kleinsten Teilgraphen genutzt wird, die ein Anfragemuster enthalten. Dazu wird die Anfrage in eine Menge von Bedingungen (*engl. constraints*) überführt. Anhand zweier Regeln können dann irrelevante Knoten in dem Indexbaum ausgeschlossen werden. Die für eine Anfrage relevanten Einträge zeigen dann in eine die Tripel enthaltende Hash-Map.

Dieser Ansatz von Udrea ist am ehesten mit den Basis-Zugriffsstrukturen des RCS vergleichbar. Wie der Subjekt-, Prädikat- und Objektindex erlaubt Udreas Verfahren das Einschränken der Kandidaten für die Beantwortung einer Anfrage. Dieser Index ist insofern nicht mit dem in Abschnitt 3.2 definierten Speichermodell kompatibel, als dass der Ansatz von einer tripelbasierten Verwaltung der RDF-Daten ausgeht; im RCS werden die Tripel dahingegen nach Subjekten gruppiert auf Datenseiten gespeichert.

4.6.3 Pfadbasierte Ansätze

Bei pfadbasierten Ansätzen werden aus der Datenbasis vorher definierte Pfade extrahiert und in einem Index gespeichert. Dies können beispielsweise alle vorkommenden Pfade, Pfade (bis zu) einer bestimmten Länge oder die häufigsten Pfade sein. Zum Beantworten einer Anfrage werden zunächst die darin enthaltenen, indexierten Pfade bestimmt und anschließend über den Index die passenden Teile des Graphen ermittelt und als Ergebnis zurückgegeben.

DataGuides. DataGuides bilden eine prägnante und genaue strukturelle Zusammenfassung von semistrukturierten Datenbanken, die jeden darin enthaltenen Pfad – und nur die – widerspiegelt [GW97]. Damit wirken sich Änderungen an der Datenbank auch auf den zugehörigen DataGuide aus. In einem DataGuide tritt ein Pfad von Kantenbezeichnern genau einmal auf. Eine Voraussetzung für einen DataGuide ist, dass die indexierte Datenbank genau ein Wurzelobjekt besitzt. Dadurch ergibt sich für den DataGuide ebenfalls eine Baumstruktur. Obwohl die Konstruktion eines DataGuide für Daten in Baumstruktur in lineare Speicherplatz- und Zeitkomplexität möglich ist, steigt im schlimmsten Fall die Komplexität für graphförmige Daten ins Exponentielle, insbesondere wenn sie Zyklen enthalten [GW99]. Durch Aufgabe der Bedingung, dass alle Pfade in der Datenbank existieren müssen, entstand der Approximate DataGuide (ADG). Da der ADG nunmehr auch nicht-existente Pfade enthalten kann, ist dieser nicht mehr als exakter Index einsetzbar. Darüber hinaus berichten die Autoren für größere Datenbanken, dass bei der Konstruktion des ADG die Pfade „gigantische Längen“ erreichen, bevor wiederkehrende Abschnitte erkannt werden.

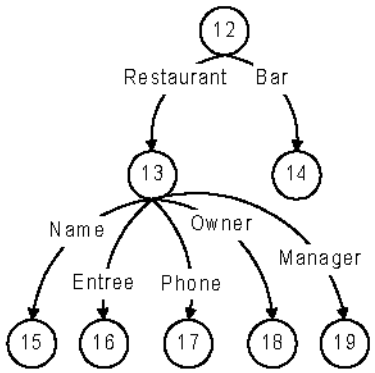


Abbildung 4.18: Beispiel eines DataGuide [GW97]

T-Index. In [MS99] betrachten Milo und Suciu Template-Indexe (T-Indexe), mit denen Pfadausdrücke mit Platzhaltern und Formeln an beliebigen Positionen beantwortet werden können, beispielsweise `(*.Restaurant) x (Menu.*.Lasagna) y`, wobei `x` und `y` Variablen bezeichnen. T-Indexe können durch Verwendung von Simulation bzw. Bisimulation zeiteffizienter als DataGuides aufgebaut werden und der benötigte Speicherplatz hängt von der Mächtigkeit der unterstützten Pfadausdrücke ab (je mächtiger die Ausdrücke, desto mehr Speicherplatz). Beispielsweise unterstützt der 1-Index Anfragen nach denjenigen Knoten, die von der Wurzel aus über einen gegebenen Pfadausdruck erreichbar sind. Dieses Verfahren sowie darauf aufbauende Arbeiten wie beispielsweise der A(k)-Index [KSBG02], der Adaptive Path Index [MCS05] und der D(k)-Index [CLO03] lassen sich auf RDF-Datenbanken nicht anwenden, da die Indexe einen Ursprungsknoten (z. B. Wurzel eines Baumes) voraussetzen.

GraphGrep. Die Herausforderung bei der Indexierung einer Graphdatenbank besteht darin, aus einer größeren Menge kleinerer Graphen die zu eine Anfrage passenden zu ermitteln. Ein darauf ausgerichtetes Indexierungsverfahren ist GraphGrep [GS02], das zu jedem Graphen in einer Datenbank alle enthaltenen Pfade bis zu eine vorher definierten Länge ermittelt. Auf Grundlage der Knotenbezeichner in einem Pfad wird ein Hashwert berechnet, der in einem Index zu den jeweiligen Graphen in Verbindung gesetzt wird. Um die Graphen zu ermitteln, die einen Anfragegraphen exakt enthalten, werden aus dem Anfragegraphen ebenfalls alle Pfade bestimmt und deren Hashwert im Index nachgeschlagen. Der Indexzugriff liefert als Ergebnis eine Obermenge der Lösungen für die Anfrage, in einem nachgelagerten Arbeitsschritt muss die Enthaltensein-Beziehung zwischen Anfragegraph und Lösungsgraphen validiert werden. Diese Indexierungsstrategie ist somit auf das Auffinden eines Graphmusters in vielen kleineren Graphen ausgerichtet. Wenn die RDF-Daten aus kleineren, gering vernetzten RDF-Graphen (z. B. FOAF-Profile) beständen oder ein RDF-Graph geeignet zerlegt werden kann, ist dieses Verfahren für das Indexieren von RDF-Daten geeignet sein. Da in dieser Arbeit die RDF-Daten nicht aus kleinere, gering vernetzten RDF-Graphen bestehen, sondern die Tripel bzgl. des Subjekts gruppiert auf Datenseiten gespeichert werden, ist das Verfahren für den RCS ungeeignet. Über Prädikat- und Objektindex werden die Tripelgruppen gut indexiert.

Access Support Relation. Im Gegensatz zu RDF-Datenbanken basieren objektorientierten Datenbanken auf einem bekannten Schema und im Gegensatz zu relationalen Datenbanken ist der Zugriff auf Objekte über eine Kette von Referenzen (funktionale Joins) wichtiger als ein Join über Attributwerte. In der Literatur finden sich Indexstrukturen, die assoziative Suchanfragen, wie die Frage nach der Existenz einer Verbindung zwischen

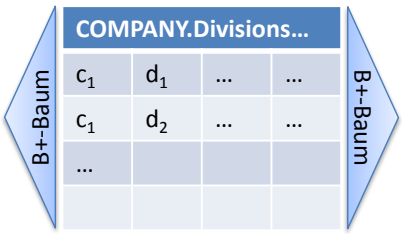


Abbildung 4.19: Speicherschema einer Access Support Relation [KM92]

zwei Objekten, und Pfadausdrücke unterstützen. Das von [KM92] entwickelte Indexierungsverfahren, *Access Support Relation*, unterstützt insbesondere den Zugriff auf Pfade von beiden Endpunkten aus, wobei sich der Pfad über mehrere Objekte hinweg erstrecken kann. Das Verfahren basiert auf dem Speichern aller Pfade einer festgelegten Länge, die von einem Objekttyp ausgehen, z. B.

```
COMPANY.Divisions.Manufactures.Composition.Name
```

Die optimale Access Support Relation wird anhand von domänenspezifischen, statistischen Informationen wie der Wahrscheinlichkeit von Anfragen und Aktualisierungsoperation bestimmt (*engl. frequently referenced class pairs*).

Letztendlich stellt dieses Verfahren eine Vorberechnung der Joins der Instanzen der beteiligten Objekttypen dar. Dazu werden zunächst temporäre binäre Relationen berechnet, die anschließend wieder miteinander verkettet werden. Indem dabei zusätzlich die jeweiligen left-outer, right-outer und full-outer Joins berechnet werden, kann eine Access Support Relation auch dann herangezogen werden, wenn in einer Anfrage nur ein Teil eines Pfades auftritt. Hierzu dienen auch linksseitig bzw. rechtsseitig gruppierte B-Bäume über der Access Support Relation (vgl. Abbildung 4.19).

Join-Index-Hierarchie. Han et al. griff in [HXF99] die Idee der Access Support Relation auf und entwickelte diese zu den so genannten Join Index Hierarchy weiter. Zum einen wird der durch die Access Support Relation belegte Speicherplatz und zum anderen der Aufwand beim Aktualisieren des Indexes adressiert. Anstatt sich auf einen zu indexierenden Pfad mit einem definierten Anfang und Ende zu fokussieren, bildet eine Join Index Hierarchy ausgehend von binären Join-Indexen auch weitere Join-Möglichkeiten ab. Beispielsweise würden mit einer kompletten Join-Index-Hierarchy auch alle Teilpfade mit abgedeckt. Durch das Entfernen von intermediären Join-Indexen kann der Aufwand für Konstruktion und Aktualisierung der Join Index Hierarchy reduziert werden, jedoch werden dann nur noch die vordefinierte Zugriffspfade unterstützt.

Obwohl die Idee der Join Index Hierarchies in einem Beitrag von Stuckenschmidt et al. [SVHB04] auch auf verteilte RDF-Datenquellen im Semantic Web übertragen worden ist, wurde der Ansatz für nicht-verteilte RDF-DBMS nicht weiter verfolgt. Eine Ursache hierfür kann darin gesehen werden, dass im Gegensatz zu OO-Datenbanken bei RDF-Daten im Allgemeinen nicht von einem fest definierten Schema ausgegangen werden kann. Dadurch ist auch die Anzahl der möglichen Objekttypen nicht begrenzt bzw. der Objekttyp einer Ressource nicht von vornherein bekannt, insbesondere könnte dieser einfach `rdf:Resource` sein.

Pfadbasierte relationale RDF-DB. In dem von Matono et al. [MAYU05] entwickelten Ansatz wird der RDF-Graph in sechs Teilgraphen zerlegt, fünf basierend auf RDF(S)-Eigenschaften (z.B. `rdf:type` und einer für die übrigen Tripel. Darüber hinaus werden für häufig angefragte Pfade extrahiert und indexiert. Für das Indexieren wird ein Nummerierungsschema verwendet, in dem jeder Knoten in einem DAG eine Pre- und Post-Order Zahl zugeordnet wird. Die Pfadinformationen werden in einer relationalen Datenbank gespeichert. Der in dieser Arbeit beschriebene Indexierungsansatz speichert die Da-

ten ebenfalls in einem RDBMS, jedoch können anstelle von Pfaden beliebige Graphmuster indexiert werden.

4.6.4 Inhaltsbasierte Indexierung

Neuere Indexierungsverfahren konzentrieren sich auf bestimmte Teilaspekte der Verarbeitung von SPARQL-Anfragen. In diesem Abschnitt werden zwei Verfahren beschrieben: Der erste indexiert die RDF-Graphen mit Hilfe von Verfahren aus dem Information Retrieval und das zweite fokussiert auf reguläre Ausdrücke.

Random Indexing. In [DPL+12] stellen die Autoren ein Verfahren vor, das auf Techniken des Information Retrieval basiert: Random Indexing, einem auf gemeinsamen Auftreten von Worten basierendes Verfahren. Zunächst wird eine Menge von virtuellen Dokumenten für einen repräsentativen Teilgraph zu einer IRI erstellt, der so genannte Kontext dieser IRI. Dieser Teilgraph enthält alle Pfade der Länge N beginnend bei der betrachteten IRI. Wenn das letzte Objekt O_N kein Literal ist, werden die Tripel $(O_N P_{N+1} L_J)$ hinzugenommen, wobei L_J ein Literal ist. Darüber hinaus werden Eigenschaften wie `rdf:type` an bestimmten Positionen in den Pfaden ausgeschlossen. Aus den Literalen werden außerdem noch vorverarbeitet, zum Beispiel werden Stopworte herausgefiltert. Ein solcher Index unterstützt vor allem Anfragen nach ähnlichen Begriffen und Dokumenten, zum Beispiel „Finde eine repräsentative IRI für einen gegebenen Begriff." Die Autoren sehen diesen Index als eine Ergänzung für die SPARQL-Anfragebearbeitung.

Reguläre Ausdrücke. Das Wiederfinden von Informationen in einem RDF-Graphen wird schwieriger, wenn die enthaltenen Ressourcen, Eigenschaften und Werte dem Nutzer unbekannt sind. In solchen Fällen ist das Verwenden von regulären Ausdrücken in SPARQL-Anfragen nützlich. In [LPL+10] beschreiben die Autoren einen auf *ngram* basierenden Index. der auf die Beantwortung von regulären Ausdrücken spezialisiert ist. Für Subjekt, Prädikat und Objekt jedes Tripels werden die ngram bestimmt, die selektivsten davon (selten und nicht redundant) werden in den Index eingefügt. Für das Speichern der ngram wird eine invertierter Index herangezogen, wobei das ngram der Schlüssel und die Liste der Vorkommen der hinterlegte Wert ist. Insgesamt werden drei Indexe angelegt, für jede Komponente eines Tripels einen.

4.7 Zusammenfassung

In diesem Kapitel wurden nutzerdefinierte Indexe über einem RDF-Graph untersucht. Ein Index wurde dabei auf Basis eines Basis-Graphmuster und dessen Lösungen definiert. Um aus einer gegebenen Menge von Indexen die für eine Anfrage nutzbaren Indexe ermitteln zu können, wurden die Beziehungen zwischen Graphmustern und ihren Lösungen untersucht. Im Ergebnis konnte nachgewiesen werden, dass ein Vorkommen des Indexmusters im Graphmuster eine notwendige und hinreichende Bedingung dafür ist. Darauf aufbauend wurde die Menge der zulässigen Indexe definiert.

Weiterhin wurde ein Speichermodell entwickelt, bei dem die Lösungen des Indexmusters als Tabelle interpretiert werden und somit ein relationales DBMS zur Verwaltung der Indexeinträge genutzt werden. Damit kann auf dessen Join-Algorithmen bei dem Zugriff auf überlappenden Indexe zurückgegriffen werden. Darüber hinaus wurde auch aufgezeigt, inwiefern neben dem sequentiellen Lesen der Indexeinträge auch das zeitgleiche Auswerten von Filterausdrücken erfolgen kann.

Kapitel 5

Anfragebearbeitung

Nachdem im vorherigen Kapitel ein natives Speichermodell für das Verwalten von RDF-Daten definiert worden ist, wird im Folgenden die Bearbeitung von Anfragen in dem System betrachtet. Für das Extrahieren von Daten aus einem RDF-Graph wurde vom W3C die Anfragesprache SPARQL [PS13] entwickelt. Da es sich bei SPARQL um eine deklarative Anfragesprache handelt, muss das Datenbankmanagementsystem die Aufgabe übernehmen, einen effizienten Ausführungsplan zu erstellen. Dazu analysiert der RDF-Anfrageprozessor (vgl. Abbildung 5.1) die Anfrage und generiert eine Reihe von Ausführungsplänen (*engl. query execution plans (QEPs)*). Zum Generieren der Varianten werden Transformationsregeln, verschiedene Algorithmen für das Ausführen von Operatoren, statistische Informationen und Indexe herangezogen. Die so generierten QEPs werden von dem Anfrageoptimierer, einer Teilkomponente des Anfrageprozessors, analysiert und schätzt deren Kosten mittels einer Kostenfunktion ab. Aus der Menge der generierten Ausführungspläne wählt der Anfrageoptimierer den kostengünstigsten aus, der danach von der Anfrageausführungskomponente des Anfrageprozessors ausgeführt wird.

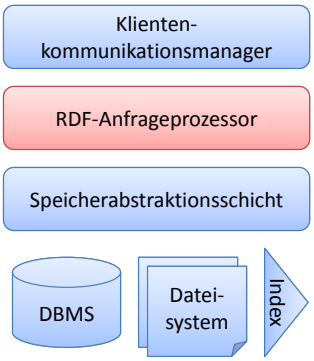


Abbildung 5.1: Generische Architektur eines RDF-Managementsystems

In diesem Kapitel wird die Verarbeitung von Anfragen bezüglich des in dieser Arbeit entwickelten nativen Speichermodells näher beschrieben. SPARQL in der Version 1.1 beinhaltet gegenüber der Vorgängerversion einige neue Sprachkonstrukte wie beispielsweise Unteranfragen, Eigenschaftspfade sowie Gruppierungen und Aggregationen. Da sternförmige Anfragen den Fokus für die Evaluation des in dieser Arbeit entwickelten Speichermodells bilden, berücksichtigt dieses Kapitel im Wesentlichen die Verarbeitung der grundlegenden Sprachkonstrukte Graphmuster, Filterausdrücke, `OPTIONAL` und `UNION`. Darüber hinaus werden im Folgenden vor allem Anfragen vom Typ `SELECT` betrachtet, da unabhängig von der Anfrageart zunächst die Vorkommen der relevanten Teilgraphen im Datengraphen und anschließend die Variablenbin-

dungen bestimmt werden müssen. Demzufolge bilden diese die Grundlage für alle anderen Anfragetypen (`ASK`, `DESCRIBE` und `CONSTRUCT`) bildet. Falls sich für einzelne Sprachkonstrukte von SPARQL 1.1 oder die anderen Anfragearten geeignetere Ausführungsstrategien ergeben, wird darauf gesondert hingewiesen.

Nach einer kurzen Einführung in die Problematik der Anfrageoptimierung in Abschnitt 5.1 wird ein Modell für die interne Repräsentation einer SPARQL-Anfrage eingeführt (Abschnitt 5.2). Im Abschnitt 5.3 werden auf Basis dieses Modells Regeln beschrieben, mit denen eine SPARQL-Anfrage im Rahmen der logischen Optimierung in eine logisch äquivalente transformiert werden kann. Anschließend werden Ausführungsstrategien spezifiziert, mit denen die Ergebnisse von Operatoren bzw. Operatorkonstellationen in einem SQGM auf Grundlage des RCS-Speichermodells berechnet werden können (Abschnitt 5.4). Im Abschnitt 5.5 werden Transformationsregeln und Ausführungsstrategien in Heuristiken miteinander kombiniert, um einen möglichst effizienten Ausführungsplan zu generieren. Die Bewertung der generierten Ausführungspläne erfolgt anhand eines Kostenmodells. Da beim RCS die Daten auf dem Sekundärspeicher gehalten werden, basiert das Kostenmodell auf den zugegriffenen Datenseiten für die Berechnung der Lösungen einer Anfrage. Die verwendeten Ansätze werden in Abschnitt 5.6 beschrieben. Nach einer Diskussion der verwandten Arbeiten im Abschnitt 5.7 werden die Erkenntnisse des Kapitels im Abschnitt 5.8 zusammengefasst.

5.1 Problembeschreibung

Gegeben den Resource Centered Store und eine SPARQL-Anfrage ist die Problemstellung, einen effizienten Ausführungsplan zu finden. Dazu wird für die Anfrage in einen algebraischen Ausdruck übersetzt, der anschließend als ein Operatorbaum dargestellt werden kann. Jeder Knoten in diesem Baum repräsentiert dabei einen algebraischen Ausdruck. Dieser Operatorbaum bildet die Grundlage für das Generieren eines effizienten Ausführungsplanes. Der Optimierer kann zum einen Transformationsregeln auf den Operatorbaum anzuwenden, die diesen in einen semantisch äquivalenten überführen. Zum anderen stehen für das Ausführen eines algebraischen Ausdrucks unter Umständen unterschiedliche Algorithmen zur Verfügung und der Optimierer kann zwischen diesen auswählen.

Um das Problem des Findens eines optimalen Ausführungsplans zu lösen, müssen vergleichbar zu relationalen DBMS die folgenden Themen betrachtet werden [Ull80]:

- Der *Suchraum* für das Generieren unterschiedlicher Anfragepläne wird durch die semantisch äquivalenten algebraischen Ausdrücke aufgespannt.

- Eine *Kostenfunktion* wird benötigt, um für jeden generierten Ausführungsplan den benötigten Aufwand (z. B. zugegriffene Datenseiten oder CPU-Zyklen) abschätzen und diesen mit anderen vergleichen zu können.

- Ein *Enumerierungsalgorithmus für Ausführungspläne* erlaubt ein Durchlaufen und auch Einschränken des Suchraumes.

Im Folgenden werden die zuvor beschriebenen Themen im Kontext des Resource Centered Store näher betrachtet und die Verfahren zum Ermitteln eines kostengünstigen Ausführungsplans beschrieben.

5.2 SPARQL Query Graph Model

Die Basis für die Optimierung von Anfragen in Datenbanksystemen bildet deren interne Repräsentation als Operatorbaum. Für das SQL-Datenbanksystem Starburst wurde von Haas et al. ein Modell zur Darstellung und Optimierung von SQL-Anfragen entwickelt, das Query Graph Model [HCL+90]. Dieses wurde unter anderem mit dem Ziel entwickelt, um die Vielzahl an möglichen Transformationsregeln auf Anfragen zu unterstützen. Zu den zu unterstützenden Strategien gehören beispielsweise das Eliminieren von redundanten Join-Anfragen, Zusammenführen von Prädikaten oder das Ersetzen von View-Definitionen. Insbesondere sollte das Anfragemodell offen für zukünftige Entwicklungen im Bereich der Anfrageoptimierung sein.

Da sich die Optimierung und Verarbeitung von SPARQL-Anfragen noch immer in der Entwicklung befindet, ist ein flexibles und erweiterbares Anfragemodell wie das Query Graph Model auch im Kontext von RDF-Datenbanksystemen wünschenswert. Im Rahmen dieser Arbeit wurde daher in Anlehnung an dieses Modell das SPARQL Query Graph Model (SQGM) für die Optimierung von SPARQL-Anfragen entwickelt. [Hee06, HH07] Neben der Analyse und Transformation von SPARQL-Anfragen erlaubt dieses Modell auch das Speichern von Metadaten (z. B. Selektivitätswerte und Kardinalitäten) an den einzelnen Operatoren.

5.2.1 Definition des SQGM

Bei einem SQGM handelt es sich um einen gerichteten Graphen, wobei ein Knoten einen Operator einer Anfrage und eine Kanten den Datenfluss zwischen Operatoren darstellt.

5.2.1.1 Operatoren

Ein Operator führt Berechnungen über eine Menge von Eingabedaten aus und generiert dabei Ausgabedaten. Die Ausgabe eines Operators ist entweder eine Menge von RDF-Tripeln, eine Multimenge von Lösungsabbildungen oder ein Wahrheitswert. Darüber hinaus können einem Operator Annotationen zugeordnet sein (z. B. *optional*). Daraus ergibt sich die folgende Definition:

Definition 5.1 (Operator)**.** *Ein* Operator op *ist als ein Tupel* (O, A) *definiert, wobei* $O \subseteq \mathcal{T} \vee O \subseteq \Omega \vee O \in \{wahr, falsch\}$ *das Ergebnis des Operators und* A *eine Menge von Attributen ist.*

Im Folgenden dient das Symbol OP als Notation für eine Menge von Operatoren. Darüber hinaus wird die Notation $\mathsf{op}.a$ für $\mathsf{op} = (O, A)$ und $a \in A$ verwendet, um den Wert des Attributs a zu referenzieren.

Die möglichen Attribute A für einen Operator hängen von dem jeweiligen Operatortyp ab. Beispielsweise hat ein Graph-Operator das Attribut *iri*,

um den zugegriffenen Graphen zu referenzieren. Im Folgenden wird ein Überblick über die definierten Operatortypen gegeben und deren Attribute sowie Ein- und Ausgaben spezifiziert. Alle Operatoren haben das Attribut *type*, anhand dessen der Typ eines Operators beispielsweise in Transformationsregeln geprüft werden kann.

Es lassen sich entsprechend ihrer Ausgabe zwei Hauptgruppen von Operatoren unterscheiden. Die erste Gruppe besteht aus den Operatoren, die eine Multimenge bzw. eine Sequenz von Lösungsabbildungen zurückliefert. Die zweite Gruppe beinhaltet die Operatoren, die eine Menge von Tripeln als Ausgabe hat. Grundsätzlich gibt es noch die Gruppe der Wahrheitswert-erzeugenden Operatoren (`ASK`), die jedoch im Folgenden nicht weiter berücksichtigt wird.

Ergebnisproduzierende Operatoren Operatoren dieses Typs erzeugen das finale Ergebnis einer Anfragen. Jede Anfrage beinhaltet nur einen einzigen Operator dieses Typs. Zu diesen gehören die folgenden Operatoren:

Ein *Select-Operator* ($\Omega \rightarrow \Omega$ bzw. $\Psi \rightarrow \Psi$) erzeugt in Abhängigkeit von den Eingabedaten entweder eine Multimenge oder eine Sequenz von Lösungsabbildungen, wobei in diesen nur Variablenbindungen für die spezifizierten Variablen enthalten sind.
type = Select
vars – berücksichtigte Variablen

Mit dem *Ask-Operator* ($\Omega \rightarrow \{\texttt{wahr}, \texttt{falsch}\}$) wird überprüft, ob dessen Eingabe eine leere Menge ist.
type = Ask

Der *Construct-Operator* ($\Omega \rightarrow \mathcal{T}$) nimmt eine Multimenge von Lösungsabbildungen als Eingabe und wendet diese auf ein Template an, um eine Menge von Tripeln zu generieren.
type = Construct
template – Template für das Generieren von Tripeln

Der *Describe-Operator* ($\Omega \rightarrow \mathcal{T}$) extrahiert aus einem RDF-Graph alle Tripel, die für die zu beschreibende Resource relevant sind.
type = Describe

Graph-Zugriffsoperatoren Graph-Zugriffsoperator dienen dem Zugriff auf RDF-Graphen oder dem Erzeugen des Default-Graphen, auf denen eine Anfrage ausgeführt wird. Zu dieser Klasse von Operatoren gehört neben dem Graph-Operator auch der Graph-Merge-Operator. Dieser dient dem Vereinigen von RDF-Graphen und somit dem Erzeugen des Default-Graphen.

Mit dem *Graph-Operator* ($\mathcal{T} \rightarrow \mathcal{T}$) werden die Tripel eines RDF-Graphen anderen Operatoren zur Verfügung gestellt, indem dessen Tripel beispielsweise von dem Sekundärspeicher gelesen werden.
type = Graph
iri – IRI eines RDF-Graphen, über die eine Anfrage ausgeführt wird
default – kennzeichnet, ob der Graph zum Default-Graph gehört

Der *Graph-Vereinigungsoperator* ($\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$) erzeugt entsprechend der Semantik für das Zusammenführen von RDF-Graphen aus den Tripelmenge

zweier RDF-Graphen einen neue Tripelmenge.
type = Merge

Graphmuster-Operatoren Während die zuvor beschriebenen Operatoren entweder als Wurzel oder als Blatt in einem Operatorbaum auftreten, kommen die nachfolgenden Operatoren als innere Knoten vor. Diese dienen vor allem der Berechnung von Graphmustern, zu denen auch Vereinigungs- und Join-Operationen gehören.

Jeder der in diesem Teil beschriebene Operator hat das Attribut *vars*, dass die Variablen benennt, für die der Operator Lösungsabbildungen generiert.

Der *Basis-Graphmuster-Operator* ($\mathcal{T} \rightarrow \Omega$) generiert eine Multimenge von Lösungsabbildungen, mit denen ein gegebenes Muster in einer Tripelmenge vorkommt. Dabei muss eine Lösungsabbildung die gegebenen Bedingungen erfüllen, um zur Ausgabe dazuzugehören.
type = BGP
pattern – zu findendes Graphmuster

Ein Filter-Operator ($\Omega \rightarrow \Omega$) berechnet für die Elemente einer Multimenge von Lösungsabbildungen den Wert eines booleschen Ausdrucks. Es werden nur die Lösungsabbildungen ausgegeben, für die dieser Wert wahr ist.
type = Filter
expression – die von den Lösungsabbildungen zu erfüllenden Bedingungen

Der *Join-Operator* ($\Omega \times \Omega \rightarrow \Omega$) verknüpft zwei Mengen von Lösungsabbildungen und generiert eine neue, indem kompatible Lösungsabbildungen zusammengeführt werden (vgl. Abschnitt 2.10).
type = Join
optional – Variablenbindungen des rechten Operands sind optional (Left-Outer-Join)

Der *Vereinigungsoperator* ($\Omega \times \Omega \rightarrow \Omega$) verknüpft ähnlich zwei Multimengen von Lösungsabbildungen, indem diese vereinigt werden.
type = Union

Lösungsmodifikationsoperatoren Die Multimenge der Lösungsabbildungen kann durch Operatoren dieser Klasse weiter verändert werden. Dazu gehört das Erzeugen einer sortierten Liste von Lösungsabbildungen und das Auswählen einer Untermenge von Lösungsabbildungen.

Jeder der in diesem Teil beschriebene Operator hat das Attribut *vars*, dass die Variablen benennt, für die der Operator Lösungsabbildungen generiert.

Der *Sortier-Operator* ($\Omega \rightarrow \Psi$) konvertiert die Multimenge der Lösungsabbildungen in eine Sequenz von Lösungsabbildungen, wobei diese entsprechend der Sortierkriterien geordnet werden. Falls das Attribut *orderBy* undefiniert ist, wird keine Veränderung der Reihenfolge der Lösungsabbildungen vorgenommen.
type = Sort
orderBy – Definition einer Ordnung auf den Lösungsabbildungen

Der *Distinct-Operator* ($\Psi \rightarrow \Psi$) eliminiert alle Duplikate in einer Sequenz von Lösungsabbildungen.
type = Distinct

Der *Auswahl-Operator* ($\Psi \rightarrow \Psi$) erzeugt eine Teilsequenz von Lösungsabbildungen beginnend bei der durch das Attribut *offset* bestimmten Lösungsabbildung. Das Attribut *limit* gibt die Anzahl der ausgegebenen Lösungsabbildungen an.

type = Slice
limit – Einschränken der Anzahl der Lösungsabbildungen im Ergebnis
offset – erste auszugebende Lösungsabbildung

5.2.1.2 Iterator

Um den Datenfluss zwischen zwei Operatoren zu beschreiben, wird im Folgenden der Begriff *Iterator* verwendet. Ein Iterator ermöglicht einem anderen Operator den Zugriff auf die Ausgabe O eines Operators $\mathsf{op} = (O, A)$. In einem Iterator ist definiert, von welchen Variablen die Lösungsabbildungen weitergegeben werden.

Definition 5.2 (Iterator)**.** *Ein* Iterator it *ist ein Tupel* $(\mathsf{op}_{src}, \mathsf{op}_{dest}, V)$*, wobei* op_{src} *den Daten liefernden Operator und* op_{dest} *den Daten verarbeitenden Operator bezeichnen.* $V \subset \mathbb{V}$ *ist die Menge der Variablen, zu deren Lösungsabbildungen der Iterator Zugriff gewährt.*

Die Menge V eines Iterators ist genau dann leer, wenn der Operator op_{src} keine Lösungsabbildungen generiert (z. B. Graph-Zugriffsoperatoren).

Eine Menge von Iteratoren wird im Folgenden mit dem Symbol IT referenziert. Der Daten liefernde Operator op_{src}wird als *Datenproduzent* und und der Daten verarbeitende Operator op_{dest} als *Datenkonsument* bezeichnet. Im Folgenden werden die Notationen src(op) für die Menge aller Datenproduzenten und dest(op) für die Menge aller Datenkonsumenten eines Operators op verwendet.

Insbesondere erlaubt die Definition des Anfragemodells, dass zwei Iteratoren denselben Operator als Datenproduzenten haben. Über die Iteratoren kann darüber hinaus spezifiziert werden bzw. dem Anfragemodell entnommen werden, auf welche Variablen ein Datenkonsument zugreift. Damit kann die Wiederverwendung der Ausgabe eines Operators ausgedrückt werden.

5.2.1.3 SPARQL Query Graph Model

Mit der Definition von Operatoren und Iteratoren ist es nun möglich, den SQGM zu definieren.

Definition 5.3 (SQGM)**.** *Ein SPARQL Query Graph Model ist eine Repräsentation einer SPARQL-Anfrage Q in Form eines Graphen. Dieser ist durch das Tupel* $(\mathsf{OP}, \mathsf{IT}, r, dflt, NG)$ *definiert, wobei*

- OP – *Menge aller aus Q resultierenden Operatoren*
- IT – *Menge aller aus Q resultierenden Iteratoren*
- r – *der ergebnisproduzierende Operator (*`SELECT`*,* `ASK`*,* `DESCRIBE` *oder* `CONSTRUCT`*)*
- $dflt$ – *ein Graph-Operator, der den Default-Graphen zur Verfügung stellt*
- NG – *eine Menge von Graph-Operatoren, die den Zugriff auf benannte RDF-Graphen ermöglichen*

5.2.2 Graphische Repräsentation

Ein SQGM hat auch eine graphische Repräsentation, in der Operatoren durch Rechtecke und Iteratoren durch Pfeile dargestellt werden. Ein Operator besteht aus einem Kopf und einem Rumpf. Der Kopf eines Operators zeigt den Typ des Operators sowie die Variablen, deren Lösungsabbildungen zur Verfügung gestellt werden. Der Rumpf beinhaltet die Attribute des Operators. Im Falle eines Graphmuster-Operators ist dies beispielsweise das zugehörige Graphmuster. Ein Iterator wird durch einen Pfeil vom datenproduzierenden zum datenkonsumierenden Operator dargestellt. Der Pfeil wird mit der Menge der Variablen annotiert, deren Variablenabbildungen transferiert werden.

Beispiel 5.1. Abbildung 5.3 zeigt das SQGM für die Anfrage aus Abbildung 5.2, mit der alle Ressourcen und optional die zugehörende Emailadresse bestimmt werden, die mit der Ressource `ex:EinePerson` über das Prädikat `foaf:knows` verbunden sind.

```
SELECT ?pers ?mbox WHERE {
  ?pers foaf:name ?name .
  ?pers foaf:knows ?other .
  OPTIONAL {
    ?pers foaf:mbox ?mbox .
  }
  FILTER (?other = ex:EinePerson)
}
```

Abbildung 5.2: Beispielanfrage

Das unterste Rechteck in Abbildung 5.3 repräsentiert den Graphoperator, der die RDF-Daten für die Anfrage bereitstellt. Für die beiden Graphmuster der Anfrage wurden zwei Operatoren vom Typ BGP generiert. Das zu verarbeitende Graphmuster befindet sich im Rumpf (Annotation `pattern`) und die zur Verfügung gestellten Variablen im Kopf des jeweiligen Operators (z. B. `?pers` und `?mbox`). Wie die Annotationen der ausgehenden Iteratoren zeigen, werden jeweils die Bindungen aller Variablen an den nachfolgenden Operator weitergegeben. Die Lösungsabbildungen beider Operatoren werden anschließend über einen Join-Operator verknüpft. Der nachfolgende Operator filtert die Ergebnisse entsprechend des Filterausdrucks `?other = ex:EinePerson`. Obwohl der Filterausdruck die Bindungen aller Variablen zur Verfügung stellt, werden von dem ausgehenden Operator nur die Bindungen der Variablen `?pers` und `?mbox` weitergegeben. Der letzte Operator generiert das Ergebnis der Anfrage. □

5.2.3 Erzeugen eines SQGM

Bevor auf Basis des SQGM eine SPARQL-Anfrage optimiert werden kann, muss diese in einen SQGM überführt werden. Eine SPARQL-Anfrage ist entsprechend der SPARQL-Spezifikation [PS13] ein Tupel (E, DS, QF), wobei E ein

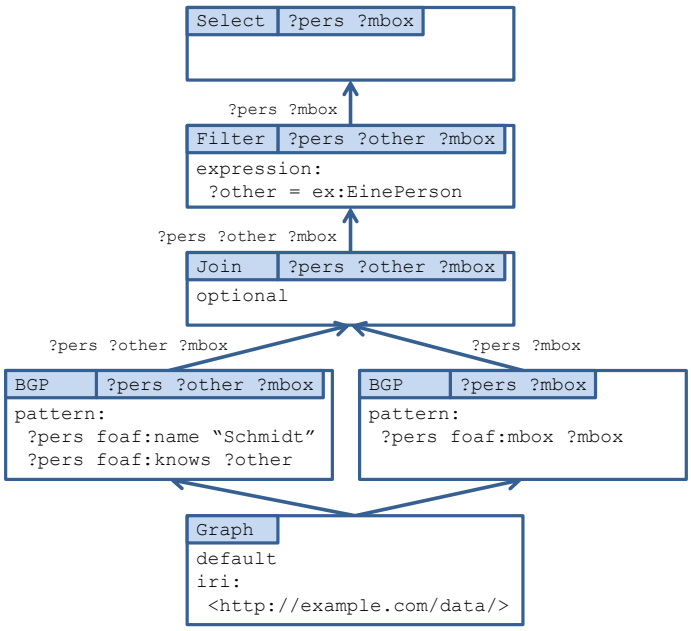


Abbildung 5.3: Graphische Darstellung des SQGM der Anfrage aus Abb. 5.2

Ausdruck der SPARQL-Algebra, DS eine RDF-Datenmenge und QF eine Ergebnisform ist. Das Überführen einer SPARQL-Anfrage in einen algebraischen Ausdruck wird in der SPARQL-Spezifikation [PS13] definiert.

Ein SQGM $(\mathsf{OP}, \mathsf{IT}, r, dflt, NG)$ wird in drei Schritten erstellt:

1. Erzeugen der Graph-Zugriffsoperatoren aus DS
2. Erzeugen der Graphmuster-Operatoren aus dem Algebraausdruck GP
3. Erzeugen des ergebnisproduzierenden Operators für die Ergebnisform R

Erzeugen der Graph-Zugriffsoperatoren Der erste Schritt besteht im Generieren des Graph-Zugriffsoperatoren für die RDF-Datenmenge DS, die aus einem Default-Graphen G und ein oder mehreren benannten Graphen $(\langle u_i \rangle, G_i)$ besteht. Dabei ist es für das Anfragemodell unerheblich, in welcher Form die RDF-Daten gespeichert werden (z. B. Hauptspeicher, Sekundärspeicher oder Web-Resource).

In einer SPARQL-Anfrage wird der Default-Graph durch eine Menge von `FROM`-Klauseln spezifiziert und die benannten Graphen durch jeweils eine `FROM NAMED`-Klausel. Beim Generieren des Operators $dflt$ werden anhand der Anzahl m an `FROM`-Klauseln die folgenden beiden Fälle unterschieden:

i) $m = 1$: Für die `FROM`-Klausel wird ein Graph-Operator op_G erzeugt und dem Attribut *iri* die in der Klausel spezifizierte IRI zugewiesen. Der Operator wird zudem mit der Annotation `DEFAULT` ausgezeichnet. Es wird $dflt = \mathsf{op}_G$ gesetzt und $\mathsf{OP} = \mathsf{OP} \cup \mathsf{op}_G$ erweitert.

ii) $m > 1$: Für jede FROM-Klausel wird wie zuvor ein Graph-Operator op_{G_i} mit dem entsprechenden Wert für das Attribut *iri* generiert. Anschließend ein Graph-Vereinigungsoperator $\mathsf{op}_\cup$ mit der Annotation DEFAULT und für jeden Graph-Operator op_{G_i} ein Iterator $\mathsf{it}(G_i, \mathsf{op}_\cup, \emptyset)$ erzeugt. Es wird $dflt = \mathsf{op}_\cup$ gesetzt und $\mathsf{OP} = \mathsf{OP} \cup \mathsf{op}_\cup$ erweitert.

In dem Anfragemodell gibt es für jeden RDF-Graphen genau einen Graph-Operator. Falls beispielsweise für des Generierens des Default-Graphen bereits ein Graph-Operator op_G für einen RDF-Graphen G erzeugt worden ist und dieser Graph auch als benannter Graph in der Anfrage vorkommt, dann wird kein weiterer Operator op_G erzeugt sondern der existierende wiederverwendet.

Erzeugen der Graphmuster-Operatoren Im anschließenden Schritt werden die Operatoren für den algebraischen Ausdruck generiert, wobei der algebraische Ausdruck entsprechend der in der Spezifikation [PS13] definierten Transformationsalgorithmus aus einer SPARQL-Anfrage generiert wird. Der entstehende algebraische Ausdruck hat bereits eine baumartige Struktur, so dass im Folgenden nur die Überführung dessen Operatoren in Operatoren des SQGM beschrieben wird. Wie bereits in der Einleitung zu diesem Kapitel erwähnt, werden dabei nur die für die Evaluation des Resource Centered Stores notwendigen Operatoren betrachten.

Bei der Überführung eines algebraischen Ausdrucks werden die Operatoren in der folgenden Reihenfolge betrachtet: Basis-Graphmuster, Graphmustergruppen[1], Filterausdrücke und Graph-Graphmuster.

Für jedes *Basis-Graphmuster* $\mathsf{BGP}(P)$ wird ein Graphmuster-Operator op erzeugt. Dabei wird der Wert von der Eigenschaft *pattern* auf P und der von *vars* auf die im Graphmuster vorkommenden Variablen $\mathrm{var}(P)$ gesetzt. Außerdem wird ein Iterator $\mathsf{it} = (dflt, \mathsf{op}, \mathsf{op}.vars)$ generiert, der den generierten Operator mit dem Operator für den Default-Graphen verknüpft. Des Weiteren wird $\mathsf{OP} = \mathsf{OP} \cup \mathsf{op}$ und $\mathsf{IT} = \mathsf{IT} \cup \mathsf{it}$ gesetzt.

Bei der Übersetzung einer *Graphmustergruppe* in einen algebraischen Ausdruck können über die Basis-Graphmuster hinaus die folgenden algebraische Ausdrücke entstehen: Join, Left Outer Join und Vereinigung. Um für den algebraischen Ausdruck einer Graphmustergruppe die Operatoren des SQGM zu erzeugen, wird davon ausgegangen, dass die Operatoren für die Argumente der jeweiligen algebraischen Ausdrücke bereits erzeugt worden sind. Für die einzelnen algebraischen Ausdrücke wird wie folgt vorgegangen:

- *Join und Left Outer Join.* Sei $\mathsf{GGP}(P_1, P_2)$ der zu transformierende algebraische Ausdruck und seien op_1 und op_2 die beiden Operatoren, die durch Übersetzen von P_1 bzw. P_2 entstanden sind. Für den Ausdruck wird ein Join-Operator $\mathsf{op}_\bowtie$ und zwei Iteratoren $\mathsf{it}_1 = (\mathsf{op}_1, \mathsf{op}_\bowtie, \mathsf{op}_1.vars)$ und $\mathsf{it}_2 = (\mathsf{op}_2, \mathsf{op}_\bowtie, \mathsf{op}_2.vars)$ erzeugt. Im Falle einer optionalen Graphmustergruppe $\mathsf{OGP}(P_1, P_2)$ wird das Attribut *optional* des Operators $\mathsf{op}_\bowtie.optional = \mathsf{it}_2$ gesetzt.

- *Vereinigung.* Sei $\mathsf{UGP}(P_1, P_2)$ der zu transformierende algebraische Ausdruck und op_1 und op_2 die beiden Operatoren, die durch Übersetzen

[1] Hierbei werden auch Vereinigungen und Verknüpfungen (Join) von Graphmustern betrachtet.

von P_1 bzw. P_2 entstanden sind. Es wird ein Vereinigungsoperator $\mathsf{op}_\cup$ sowie zwei Iteratoren $\mathsf{it} = (\mathsf{op}_1, \mathsf{op}_\cup, \mathsf{op}_1.vars)$ und $\mathsf{it} = (\mathsf{op}_2, \mathsf{op}_\cup, \mathsf{op}_2.vars)$ erzeugt.

Die Mengen OP und IT werden danach um die neu erzeugten Operatoren und Iteratoren erweitert.

Für jeden *Filter* $\mathsf{FS}(P)$ im algebraischen SPARQL-Ausdruck wird ein Filteroperator $\mathsf{op}_{\mathsf{FS}}$ der Menge der Operatoren OP des SQGM hinzugefügt. Dabei wird der Wert des Attributs *expression* auf das Prädikat des Filters FS gesetzt. Sei außerdem op der Operator, der durch das Überführen von P entstanden ist. Dann wird ein Iterator $\mathsf{it} = (\mathsf{op}, \mathsf{op}_{\mathsf{FS}}, \mathsf{op}.vars)$ der Menge der Iteratoren IT hinzugefügt.

Mit einem *Graph-Graphmuster* $\mathsf{GP}(x, P), x \in \mathbb{V} \cup \mathbb{I}$ werden die für ein Graphmuster zu verwendenden RDF-Graphen spezifiziert. Zum Überführen eines solchen Ausdrucks wird wie zuvor beschrieben zunächst der enthaltene Graphmusterausdruck rekursiv in Operatoren des SQGM überführt. Sei op der zu P korrespondierende Operator. Jedoch werden in diesem Fall Iteratoren zu Graph-Operatoren erstellt. Dabei sind zwei Fälle zu unterscheiden:

1. $x \in \mathbb{I}$: Für den Operator op wird ein Iterator $\mathsf{it} = (\mathsf{op}_{G_x}, \mathsf{op}, \emptyset)$ erzeugt.
2. $x \in \mathbb{V}$: Für den Operator op werden Iteratoren $\mathsf{it}_i = (\mathsf{op}_{G_i}, \mathsf{op}, \emptyset), \mathsf{op}_{G_i} \in NG$ zu allen Graph-Operatoren in generiert.

Anschließend werden die Mengen OP respektive IT um die neu erzeugten Operatoren und Iteratoren erweitert.

Ein *Sortier-*, *Distinct-* und *Auswahl-Operator* op wird für den jeweils korrespondierenden algebraischen Ausdruck in der SPARQL-Anfrage generiert und der Menge OP hinzugefügt. Anschließend wird zu der Menge IT ein Iterator $\mathsf{it} = (\mathsf{op}_{root}, \mathsf{op}, \mathsf{op}_{root}.vars)$ hinzugenommen. Falls es sich beim generierten Operator um einen Sortier- und/oder Auswahl-Operator handelt, werden darüber hinaus die Attribute *orderBy* bzw. *limit* und *offset* entsprechend gesetzt.

Erzeugen des ergebnisproduzierenden Operators Der letzte Schritt beinhaltet das Erzeugen des ergebnisproduzierenden Operators. Entsprechend des Anfragetyps – `SELECT`, `ASK`, `DESCRIBE` oder `CONSTRUCT` – wird ein Operator op generiert. Bei einem Select-Operators werden die Attribute *vars* der zurückzugebenden Variablenbindungen und bei einem Construct-Operator wird das Attribut *template* gesetzt.

Wie auch beim Erzeugen der Lösungsmodifikationsoperatoren gibt es in diesem Schritt nur einen Operator op_{root}, dem kein Datenkonsument zugeordnet ist. Dieser Knoten wird nun mit dem gerade generierten Knoten op verknüpft, indem ein Iterator $\mathsf{it} = (\mathsf{op}_{root}, \mathsf{op}, \mathsf{op}_{root}.vars)$ erstellt wird. Darüber hinaus werden $R = \mathsf{op}$ gesetzt und die Mengen OP und IT entsprechend erweitert.

5.3 Transformationsregeln

Das Ziel des Anwendens von Transformationsregeln auf das SQGM einer Anfragen ist die Anordnung von Operatoren so zu verändern, dass zum einen ein

semantisch äquivalentes Modell erzeugt wird und zum anderen das Ergebnis der Anfrage in kürzerer Zeit berechnet werden kann. Die Strategien ähneln der in der Einleitung zu diesem Abschnitt erwähnten Strategien für die Optimierung von SQL-Anfragen, z. B. Eliminieren von Join-Operationen oder das Zusammenführen und Vereinfachen von Filterausdrücken. Die Transformation eines SQGM wird aber auch im Kontext des Resource Centered Store betrachtet, um die inhärenten Eigenschaften des Speichermodells zu berücksichtigen. Aus diesem Grund werden auch Regeln eingeführt, die die Tripelmuster eines Basis-Graphmuster-Operator aufteilt und anstelle dessen einen Join-Operator erzeugt.

Eine wesentliche Eigenschaft von Transformationsregeln ist, dass jede von ihnen das Anfragemodell in ein semantisch äquivalentes überführt. Diese Eigenschaft wird durch die folgende Definition beschrieben:

Definition 5.4 (Semantische Äquivalenz)**.** *Seien mit* $Q = (\mathsf{OP}, \mathsf{IT}, r, dflt, NG)$ *und* $Q' = (\mathsf{OP}', \mathsf{IT}', r', dflt', NG')$ *zwei SQGM gegeben, wobei* $r = (O, A)$ *und* $r' = (O', A')$ *die ergebnisproduzierenden Operatoren sind.* Q_1 *und* Q_2 *sind* semantisch äquivalent, *falls* $O = O'$ *gilt.*

Die Gleichung $O = O'$ muss dabei im Kontext des ergebnisproduzierenden Operators betrachtet werden. Falls dieser Operator ein Select-Operator ist muss unterschieden werden, ob eine Sequenz oder eine Multimenge von Lösungsabbildungen erzeugt wird. Entsprechend muss beim Überprüfen der Gleichheit des Ergebnisses die Ordnung der Lösungsabbildungen berücksichtigt werden. Im Falle eines Construct- oder Describe-Operators werden die erzeugen Tripelmengen auf Äquivalenz überprüft, d.h. jedes Tripel der einen Menge ist in der anderen enthalten und umgekehrt. Bei einem Ask-Operator müssen die beiden Wahrheitswerte übereinstimmen.

Eine Transformationsregel besteht aus einer Menge von Bedingungen und einer Sequenz von Aktionen. Die Bedingung legt die Voraussetzungen fest, unter denen eine Transformationsregel angewendet werden kann. Falls die Voraussetzungen für eine Transformationsregel erfüllt sind, dann überführen die Aktionen den ursprünglichen SQGM Q in ein semantisch äquivalentes Anfragemodell Q'.

Definition 5.5 (Transformationsregel)**.** *Eine* Transformationsregel *T ist ein Tupel (C, A), wobei C die Voraussetzungen für die Anwendbarkeit der Regel mittels eines booleschen Prädikats beschreibt und A eine Sequenz von Instruktionen für das Modifizieren eines SQGM ist.*

Mit einer Transformationsregel wird ein Teilgraph eines SGQM in einen anderen überführt. Der ursprüngliche und der modifizierte Teilgraph haben jeweils einen Wurzelknoten op_{root} bzw. op'_{root}. Für die nachfolgend definierten Transformationsregeln wird implizit angenommen, dass die ausgehenden Iteratoren op_{root} nach Anwendung der Transformationsregel mit der Wurzel op'_{root} des neuen Teilgraphen verknüpft werden, d.h. jeder Iterator $\mathsf{it}_i = (\mathsf{op}_{root}, \mathsf{op}_i, v_i)$ wird durch einen Iterator $\mathsf{it}'_i = (\mathsf{op}'_{root}, \mathsf{op}_i, v_i)$ im SQGM ersetzt

Mit $\mathsf{var}(\mathsf{op}) \subset \mathbb{V}$ wird für einen Operator $\mathsf{op} \in \mathsf{OP}$ die Menge der Variablen bezeichnet, die durch den Operator op zugegriffen werden. Für den Filter-Operator op mit dem Ausdruck $\mathsf{op}.expression = ?a > 0$ ist beispielsweise $\mathsf{var}(\mathsf{op}) = \{?a\}$.

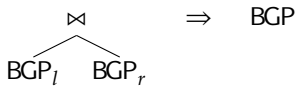
Im Folgenden wird zunächst die Transformationsregel für das Eliminieren einer Join-Operation definiert und deren Funktionsweise anhand eines Beispiels erläutert. Im Anschluss daran werden die weiteren für das SQGM definierten Transformationsregeln prägnant beschrieben.

5.3.1 Transformation von Join-Operatoren

Join-Operationen stellen einen wesentlichen Baustein von SPARQL-Anfragen dar. Im Folgenden werden Regeln zur Transformation von Join-Operatoren beschrieben. Jede Transformationsregel wird dabei eingangs der Beschreibung durch eine Illustration schematisch[2] veranschaulicht.

5.3.1.1 Eliminieren eines Join-Operators

Die erste Transformationsregel hat zum Ziel zwei Basis-Graphmuster-Operatoren zusammenzuführen, die über einen Join-Operator miteinander verbunden sind. Als Resultat des Zusammenführens der drei involvierten Operatoren entsteht ein einzelner Basis-Graphmuster-Operator. Die Transformationsregel ist wie folgt definiert.



Transformationsregel 1.
Definitionsbereich: $\{\mathsf{op}_{\bowtie} : \mathsf{op}.type = \text{Join}\}$

Voraussetzungen: Seien $\mathsf{op}_{\bowtie}$ ein Operator aus dem Definitionsbereich und $\mathsf{it}_l = (\mathsf{op}_l, \mathsf{op}_{\bowtie}, v_l)$ und $\mathsf{it}_r = (\mathsf{op}_r, \mathsf{op}_{\bowtie}, v_r)$ die beiden eingehenden Iteratoren von $\mathsf{op}_{\bowtie}$. Es gelte:

i) $\mathsf{op}_l.type = \text{BGP} \wedge \mathsf{op}_r.type = \text{BGP}$

ii) $|\text{dest}(\mathsf{op}_l)| = 1 \wedge |\text{dest}(\mathsf{op}_r)| = 1$

iii) $\mathsf{op}_{\bowtie}.optional = \text{undef}$

iv) $\text{src}(\mathsf{op}_l) = \text{src}(\mathsf{op}_r)$

Modifikation: Folgende Veränderungen werden am SQGM vorgenommen:

- op_l und op_r werden zu einem Operator op mit $\mathsf{op}.type = \text{BGP}$, $\mathsf{op}.vars = \mathsf{op}_l.vars \cup \mathsf{op}_r.vars$ und $\mathsf{op}.pattern = \mathsf{op}_l.pattern \sqcup \mathsf{op}_r.pattern$ vereinigt[3] sowie zu OP hinzugenommen.
- Für den Operator op werden Iteratoren zu den Datenproduzenten von op_l bzw. op_r und zu den Datenkonsumenten von $\mathsf{op}_{\bowtie}$ erzeugt und zu der Menge IT hinzugenommen.
- $\mathsf{op}_{\bowtie}, \mathsf{op}_l, \mathsf{op}_r$ und alle mit ihnen verbundenen Iteratoren werden aus dem SQGM entfernt.

[2] Es werden die in der Transformation involvierten Operatortypen dargestellt.

[3] Die Vereinigung von Graphmustern ist analog zur Vereinigung von RDF-Graphen definiert. Insbesondere muss beispielsweise Konflikte in unbenannte Ressourcen aufgelöst werden.

Äquivalenz: Damit durch diese Transformationsregel ein semantisch äquivalenter SQGM erzeugt wird, muss die Äquivalenz des Join-Operators mit dem erzeugten Basis-Graphmuster-Operator gezeigt werden. Die Ausführung der übrigen Operatoren in dem SQGM wird durch die Transformationsregel nicht beeinflusst, da nach Voraussetzung ii) die Operatoren op_1 und op_2 nur den Join-Operator als Datenkonsument haben.

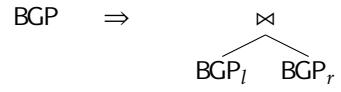
Sei mit $G \in \mathcal{G}$ ein beliebiger RDF-Graph gegeben. Seien $P_1 \in \mathsf{BGP}$ und $P_2 \in \mathsf{BGP}$ zwei Basis-Graphmuster und Ω_1 und Ω_2 die jeweils zugehörigen Multimengen von Lösungsabbildungen. Nach [PS13] ist der Join wie folgt definiert:

$$\Omega_1 \bowtie \Omega_2 = \{merge(\mu_1, \mu_2) : \mu_1 \in \Omega_1 \wedge \mu_2 \in \Omega_2 \wedge \mu_1, \mu_2 kompatibel\}$$

Zu jeder Lösungsabbildung $\mu \in \Omega_1 \bowtie \Omega_2$ gibt es also zwei kompatible Lösungsabbildungen μ_1 und μ_2 mit RDF-Instanzabbildungen σ_1 und σ_2, so dass $\mu_1(\sigma_1(P_1)) \subset G$ und $\mu_2(\sigma_2(P_2)) \subset G$ gilt. Da $\mathsf{op}.pattern = P_1 \sqcup P_2$ und μ_1 und μ_2 kompatibel sind, gilt für $\sigma = \sigma_2 \circ \sigma_1$ auch $\mu(\sigma(P)) \subset G$.

5.3.1.2 Zerlegen eines Basis-Graphmuster-Operators

Diese Transformationsregel substituiert einen Basis-Graphmuster-Operator durch einen Join von zwei Basis-Graphmuster-Operatoren. Da der Resource Centered Store auf die Evaluation von Sternmustern spezialisiert ist, kann diese Transformationsregel beispielsweise dazu genutzt werden, um Basis-Graphmuster-Operatoren zu mit einem sternförmigen zu erzeugen.



Transformationsregel 2.
Definitionsbereich: $\{\mathsf{op} : \mathsf{op}.type = \mathrm{BGP}\}$

Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich.

Modifikation: Sei P_l und P_r die Zerlegung von $\mathsf{op}.pattern$ in ein zwei Graphmuster. Folgende Veränderungen werden am SQGM vorgenommen:

- Es werden drei Operatoren $\mathsf{op}_l, \mathsf{op}_r$ und $\mathsf{op}_\bowtie$ wie folgt definiert und zu OP hinzugenommen:
  - $\mathsf{op}_l.type = \mathrm{BGP}$, $\mathsf{op}_l.pattern = P_l$ und $\mathsf{op}_l.vars = \mathrm{var}(P_l)$
  - $\mathsf{op}_r.type = \mathrm{BGP}$, $\mathsf{op}_r.pattern = P_r$ und $\mathsf{op}_l.vars = \mathrm{var}(P_r)$
  - $\mathsf{op}_\bowtie.type = \mathrm{Join}$, $\mathsf{op}_l.optional = \mathrm{undef}$ und $\mathsf{op}_l.vars = \mathsf{op}.vars$
- Es werden Iteratoren $\mathsf{it}_l = (\mathsf{op}_l, \mathsf{op}_\bowtie, \mathsf{op}_l.vars)$ und $\mathsf{it}_r = (\mathsf{op}_r, \mathsf{op}_\bowtie, \mathsf{op}_r.vars)$ erzeugt. Die Datenproduzenten src(op) werden über Iteratoren mit op_l und op_r und die Datenkonsumenten dest(op) werden mit $\mathsf{op}_\bowtie$ verknüpft.
- op und alle verbundenen Iteratoren werden aus dem SQGM entfernt.

Äquivalenz: Sei mit $G \in \mathcal{G}$ ein beliebiger Graph gegeben. Weiterhin sei $\mu \in \Omega$ eine beliebige Lösungsabbildung für das Graphmuster $P = \mathsf{op}.pattern$. Dann gibt es eine RDF-Instanzabbildung σ, so dass $\mu(\sigma(P)) \subset G$ ist. Für eine Zerlegung $P = P_1 \sqcup P_2$ lassen sich durch Einschränken des Definitionsbereichs

von μ zwei kompatible Lösungsabbildungen μ_1 und μ_2 konstruieren, so dass $\mu = merge(\mu_1, \mu_2)$ und μ_1 eine Lösungsabbildung von P_1 und μ_2 eine von P_2 ist. Damit ist μ auch eine Lösungsabbildung des Joins $P_1 \bowtie P_2$.

Anhand der beiden vorherigen Transformationsregeln wurde detailliert gezeigt, wie die Operatoren und Iteratoren eines SQGM modifiziert werden. In den folgenden Abschnitten werden die Modifikationen in verkürzter Form wiedergeben, um die Transformationsregeln prägnant darzustellen.

5.3.2 Transformation von Filter-Operatoren

Die in diesem Abschnitt beschriebenen Transformationsregeln befassen sich mit dem Verschieben, Zusammenfassen und Zerlegen von Filter-Operatoren im SQGM. Durch Anwenden dieser Regeln können beispielsweise Filterausdrücke vereinfacht oder frühzeitig ausgewertet werden.

5.3.2.1 Zusammenführen von Filter-Operatoren

Durch die beiden nachfolgenden Transformationsregeln können zwei Filter-Operatoren miteinander kombiniert werden. Falls in dem Filterausdruck Prädikate über dieselbe Variable existieren, kann dieser gegebenenfalls vereinfacht werden (z. B. `?x > 0 || ?x > 1` ⇒ `?x > 1`).

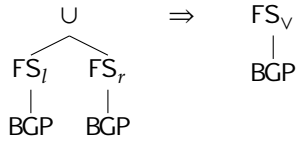
Transformationsregel 3.
Definitionsbereich: {op : op.*type* = FS}

Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich und sei it = (op, op$_{FS}$, v) der eingehende Operator von op. Es gelte op$_{\mathsf{FS}}$.*type* = FS und |dest(op$_{\mathsf{FS}}$)| = 1

Modifikation: Die beiden Operatoren op und op$_{\mathsf{FS}}$ werden durch ein Operator op′ mit op′.*type* = FS, op′.*expression* = op.*expression* ∧ op$_{\mathsf{FS}}$.*expression* und op′.*vars* = op.*vars* ersetzt.

Äquivalenz: Da die Filterausdrücke im ursprünglichen SQGM nacheinander ausgewertet werden und op$_{\mathsf{FS}}$ keinen weiteren Datenkonsumenten hat, gilt die semantische Äquivalenz des modifizierten SQGM offensichtlich.

Mit der Transformationsregel 4 können nicht nur zwei Filterausdrücke zu einem einzigen zusammengefasst werden, sondern gleichzeitig wird ein Vereinigungsoperator entfernt. Voraussetzung dafür ist, dass die beiden Filterausdrücke über demselben Basis-Graphmuster ausgewertet werden.



Transformationsregel 4.
Definitionsbereich: {op : op.*type* = Union}

Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich und seien it$_l$ = (op$_l$, op, v_l) und it$_r$ = (op$_r$, op, v_r) die eingehenden Operatoren von op. Darüber hinaus seien it$_{\mathsf{BGP}_l}$ = (op$_{\mathsf{BGP}_l}$, op$_j$, v_{BGP_l}) und it$_{\mathsf{BGP}_r}$ = (op$_{\mathsf{BGP}_r}$, op, v_{BGP_r}) die eingehenden Operatoren von op$_l$ bzw. op$_r$. Es gelte:

i) $\mathsf{op}_l.type = \text{FS} \wedge \mathsf{op}_r.type = \text{FS} \wedge \mathsf{op}_{\text{BGP}_l}.type = \text{BGP} \wedge \mathsf{op}_{\text{BGP}_r}.type = \text{BGP}$

ii) $|\text{dest}(\mathsf{op}_l)| = 1 \wedge |\text{dest}(\mathsf{op}_r)| = 1 \wedge |\text{dest}(\mathsf{op}_{\text{BGP}_l})| = 1 \wedge |\text{dest}(\mathsf{op}_{\text{BGP}_l})| = 1$

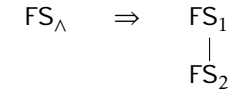
iii) $\text{src}(\mathsf{op}_{\text{BGP}_l}) = \text{src}(\mathsf{op}_{\text{BGP}_r})$

iv) $\mathsf{op}_{\text{BGP}_l}.pattern = \mathsf{op}_{\text{BGP}_r}.pattern$

Modifikation: Die fünf involvierten Operatoren und Iteratoren werden durch die Nacheinanderausführung eines Basis-Graphmuster- und eines Filter-Operators ersetzt. Diese beiden Operatoren op_{BGP} und op_{FS} sind wie folgt definiert:

- $\mathsf{op}_{\text{BGP}}.type = \text{BGP}$, $\mathsf{op}_{\text{BGP}}.pattern = \mathsf{op}_{\text{BGP}_l}.pattern$, $\mathsf{op}_{\text{BGP}}.vars = \mathsf{op}.vars$
- $\mathsf{op}_{\text{FS}}.type = \text{FS}$, $\mathsf{op}_{\text{FS}}.expression = \mathsf{op}_l.expression \vee \mathsf{op}_r.expression$

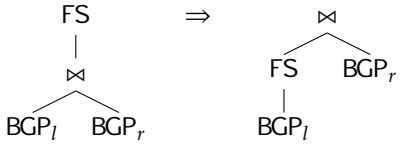
5.3.2.2 Zerlegen von Filter-Operatoren



Um die Voraussetzungen für das Verschieben von Filter-Operatoren in einem SGQM zu schaffen, kann ein Ersetzen eines Filter-Operators durch die Hintereinanderausführung zweier Filter-Operatoren erforderlich sein, die in Kombination semantisch äquivalent zu dem ursprünglichen Operator sind. Somit sind die nachfolgenden Transformationsregeln invers zu den Regeln 3 und 4.

Transformationsregel 5.
Definitionsbereich: $\{\mathsf{op} : \mathsf{op}.type = \text{FS}\}$
Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich und sei dessen Filterausdruck von der Form $\mathsf{op}.expression = e_1 \wedge e_2$.
Modifikation: Der Operator op wird durch die Nacheinanderausführung der Filter-Operatoren op_1 und op_2 mit $\mathsf{op}_1.type = \text{FS}$ und $\mathsf{op}_1.expression = e_1$ bzw. $\mathsf{op}_2.type = \text{FS}$ und $\mathsf{op}_2.expression = e_2$ ersetzt.
Äquivalenz: Die semantische Äquivalenz der beiden SQGMs gilt offensichtlich.

5.3.2.3 Verschieben von Filter-Operatoren



Wie aus der relationalen Algebra her bekannt, kann auch in der SPARQL-Algebra ein Filter-Operator vor einem Join-Operator ausgewertet werden. Die Voraussetzung hierfür ist, dass in dem Filterausdruck nur Variablen des einen eingehenden Operators verwendet werden. Es ergibt sich die folgende Transformationsregel:

Transformationsregel 6.
Definitionsbereich: $\{\mathsf{op} : \mathsf{op}.type = \text{FS}\}$
Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich. Sei $\mathsf{it} = (\mathsf{op}_{\bowtie}, \mathsf{op}, v)$ der eingehende Iterator von op sowie $\mathsf{it}_l = (\mathsf{op}_l, \mathsf{op}_{\bowtie}, v_l)$ und $\mathsf{it}_r = (\mathsf{op}_r, \mathsf{op}_{\bowtie}, v_r)$ die eingehenden Iteratoren von $\mathsf{op}_{\bowtie}$. O.B.d.A. gelte für den op_r das Folgende $\text{var}(\mathsf{op}) \cap \text{var}(op_r) = \emptyset$.

Modifikation: Ersetze die Iteratoren it durch $\mathsf{it}' = (\mathsf{op}, \mathsf{op}_{\bowtie}, v_l)$ und it_l durch $\mathsf{it}'_l = (\mathsf{op}_l, \mathsf{op}, v_l)$.
Äquivalenz: Es wird gezeigt, dass jede Lösungsabbildung für den ursprünglichen Operatorbaum auch eine für den modifizierten ist. Sei Ω die Menge der Lösungsabbildungen von $\mathsf{op}(\mathsf{op}_l \bowtie \mathsf{op}_r)$. Da $\mathsf{op}.expression$ nur Variablen enthält, die in op_l verwendet werden, hat ein Ausführen des Filterausdrucks FS vor der Join-Operation dieselbe Wirkung wie danach.

Die Transformationsregel 6 kann analog für den Fall $\mathrm{var}(\mathsf{op}) \cap v_l = \emptyset$ definiert werden.

5.3.3 Einschränken der Variablenbindungen

Durch einen Basis-Graphmuster-Operator werden die Lösungsabbildungen für alle in dem Graphmuster enthaltenen Variablen generiert. Anfragen können aber so gestaltet sein, dass später ausgeführte Operatoren nicht alle Variablenbindungen erforderlich sind. Beispielsweise werden in der Anfrage `SELECT ?s WHERE { ?s rdf:type ?o}` die Bindungen der Variable ?o für die Auswertung des Select-Operators nicht benötigt. Mit der nachfolgenden Transformationsregel werden die durch einen Iterator zugreifbaren Variablenbindungen auf die notwendigen reduziert.

Transformationsregel 7.
Definitionsbereich: $\mathsf{it} \in \mathsf{IT}$
Voraussetzungen: Sei $\mathsf{it} = (\mathsf{op}_{src}, \mathsf{op}_{dest}, v)$ ein Iterator aus dem Definitionsbereich mit $v \neq \emptyset$.
Modifikation: Die Variablen eines Iterators it ist induktiv über $(\mathsf{OP}, \mathsf{IT}, r, dflt.NG)$ definiert:

- Sei $\mathsf{op}_{dest} = r$: $v = \mathrm{var}(r)$
- Sei $\mathsf{op}_{dest} \neq r$: $v = \mathrm{var}(op_{dest}) \cup V'$, wobei $V' = \bigcup_{\mathsf{it}_i} v_i$ mit $\mathsf{it}_i = (\mathsf{op}_{dest}, \mathsf{op}_i, v_i)$

Äquivalenz: Aufgrund der Konstruktion der Menge v werden alle für die nachfolgenden Operatoren benötigten Variablenbindungen beibehalten. Darüber hinaus wird unverändert über alle vom Operator op_{src} gelieferten Lösungsabbildungen iteriert.

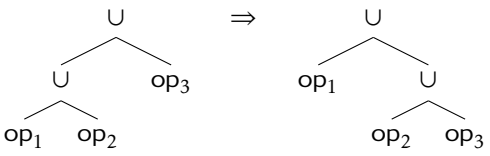
5.3.4 Beziehungen zwischen Vereinigung und Join

Für Vereinigungs- und Join-Operatoren lassen sich Gleichungen für Assoziativität, Kommutativität und Distributivität aufstellen [PAG09]. Während Kommutativität inhärenter Bestandteil des SQGMs müssen dafür keine Transformationsregeln definiert werden (Gleichungen 1 und 2), kann aufgrund der Assoziativität von Vereinigungs- und Join-Operatoren die Struktur eines SQGMs verändert werden. Beispielsweise ist es dadurch möglich, einen linksseitig tiefen in einen buschigen Vereinigungs-/Join-Baum zu überführen. Die folgenden Gleichungen lassen sich definieren:

1. $\mathsf{op}_1 \cup \mathsf{op}_2 = \mathsf{op}_2 \cup \mathsf{op}_1$

2. $\mathsf{op}_1 \bowtie \mathsf{op}_2 = \mathsf{op}_2 \bowtie \mathsf{op}_1$
3. $(\mathsf{op}_1 \cup \mathsf{op}_2) \cup \mathsf{op}_3 = \mathsf{op}_1 \cup (\mathsf{op}_2 \cup \mathsf{op}_3)$
4. $(\mathsf{op}_1 \bowtie \mathsf{op}_2) \bowtie \mathsf{op}_3 = \mathsf{op}_1 \bowtie (\mathsf{op}_2 \bowtie \mathsf{op}_3)$
5. $(\mathsf{op}_1 \bowtie \mathsf{op}_2) \cup \mathsf{op}_3 = (\mathsf{op}_1 \bowtie \mathsf{op}_3) \cup (\mathsf{op}_2 \bowtie \mathsf{op}_3)$

Die dritte Gleichung in der obigen Auflistung beschreibt die Assoziativität zwischen Vereinigungsoperatoren. Die nachfolgende Transformationsregel kann zum Verändern eines SQGMs entsprechend dieser Gleichung verwendet werden.

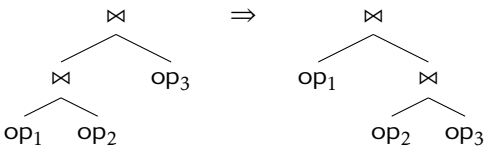


Transformationsregel 8.
Definitionsbereich: $\{\mathsf{op} : \mathsf{op}.type = \text{Union}\}$
Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich. Die Iteratoren des Operatorbaumes seien die Folgenden:

- $\mathsf{it}_\cup = (\mathsf{op}_\cup, \mathsf{op}, v_1 \cup v_2)$
- $\mathsf{it}_1 = (\mathsf{op}_1, \mathsf{op}_\cup, v_1)$
- $\mathsf{it}_2 = (\mathsf{op}_2, \mathsf{op}_\cup, v_2)$
- $\mathsf{it}_3 = (\mathsf{op}_3, \mathsf{op}, v_3)$

Modifikation: Die Iteratoren $\mathsf{it}_\cup$, it_1 und it_3 werden durch die Iteratoren $\mathsf{it}'_\cup = (\mathsf{op}_\cup, \mathsf{op}, v_2 \cup v_3)$, $\mathsf{it}'_1 = (\mathsf{op}_1, \mathsf{op}, v_1)$ beziehungsweise $\mathsf{it}'_3 = (\mathsf{op}_3, \mathsf{op}_\cup, v_3)$ ersetzt.
Äquivalenz: Siehe [PAG09]

Analog zur Transformationsregel 8 können kaskadierende Join-Operatoren eines SQGM aufgrund der Assoziativität umgeformt werden.



Transformationsregel 9.
Definitionsbereich: $\{\mathsf{op} : \mathsf{op}.type = \text{Join}\}$
Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich. Die Iteratoren des Operatorbaumes seien die Folgenden:

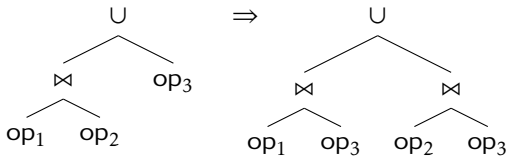
- $\mathsf{it}_\bowtie = (\mathsf{op}_\bowtie, \mathsf{op}, v_1 \cup v_2)$
- $\mathsf{it}_1 = (\mathsf{op}_1, \mathsf{op}_\bowtie, v_1)$
- $\mathsf{it}_2 = (\mathsf{op}_2, \mathsf{op}_\bowtie, v_2)$
- $\mathsf{it}_3 = (\mathsf{op}_3, \mathsf{op}, v_3)$

Modifikation: Die Iteratoren $\mathsf{it}_{\bowtie}$, it_1 und it_3 werden durch die folgenden Iteratoren $\mathsf{it}'_{\bowtie} = (\mathsf{op}_{\bowtie}, \mathsf{op}, v_2 \cup v_3)$, $\mathsf{it}'_1 = (\mathsf{op}_1, \mathsf{op}, v_1)$ beziehungsweise $\mathsf{it}'_3 = (\mathsf{op}_3, \mathsf{op}_{\bowtie}, v_3)$ ersetzt.

Äquivalenz: Siehe [PAG09]

Da Kommutativität implizit durch das Anfragemodell definiert ist, ist eine Definition der inversen Transformationsregeln von 8 und 9 nicht erforderlich, d.h., die inversen Regeln sind identisch zu den soeben definierten Regeln.

Darüber hinaus existiert zwischen Vereinigungs- und Join-Operator ein Distributivgesetz. Die zugehörige Transformationsregel wird im Folgenden beschrieben:



Transformationsregel 10.
Definitionsbereich: $\{\mathsf{op} : \mathsf{op}.type = \text{Union}\}$
Voraussetzungen: Sei op ein Operator aus dem Definitionsbereich Die Iteratoren des Operatorbaums seien die Folgenden:

- $\mathsf{it}_{\bowtie} = (\mathsf{op}_{\bowtie}, \mathsf{op}, v_1 \cup v_2)$
- $\mathsf{it}_1 = (\mathsf{op}_1, \mathsf{op}_{\bowtie}, v_1)$
- $\mathsf{it}_2 = (\mathsf{op}_2, \mathsf{op}_{\bowtie}, v_2)$
- $\mathsf{it}_3 = (\mathsf{op}_3, \mathsf{op}, v_3)$

Modifikation: Es wir ein neuer Operator $\mathsf{op}'_{\bowtie}$ mit $\mathsf{op}.type$ = Join erzeugt. Die bisherigen Iteratoren werden durch die folgenden ersetzt:

- $\mathsf{it}_{\bowtie} = (\mathsf{op}_{\bowtie}, \mathsf{op}, v_1 \cup v_3)$
- $\mathsf{it}_1 = (\mathsf{op}_1, \mathsf{op}_{\bowtie}, v_1)$
- $\mathsf{it}_2 = (\mathsf{op}_2, \mathsf{op}'_{\bowtie}, v_2)$
- $\mathsf{it}_3 = (\mathsf{op}_3, \mathsf{op}_{\bowtie}, v_3)$
- $\mathsf{it}'_3 = (\mathsf{op}_3, \mathsf{op}'_{\bowtie}, v_3)$
- $\mathsf{it}'_{\bowtie} = (\mathsf{op}'_{\bowtie}, \mathsf{op}, v_2 \cup v_3)$

Äquivalenz: Siehe [PAG09]

5.3.5 Weitere Transformationsregeln

Neben den beschriebenen eher strukturell begründeten Transformationen ist auch einen semantische Analyse der Operatoren sinnvoll, um die Evaluation von Operatoren zu vereinfachen oder komplett aus dem SQGM zu entfernen. Da die meisten dieser Analysen aus relationalen Datenbanksystemen bekannt sind (vgl. beispielsweise [BG06, CGM90]) bzw. von einfacher Struktur sind, werden auf diese hier nur in Kürze eingegangen.

- *Variablen ersetzen* – Falls in einem Filterausdruck zwei Variablen gleichgesetzt werden (z. B. `?x = ?y`), dann kann die eine durch die andere ersetzt werden. Als Ergebnis dieser Ersetzung können beispielsweise zwei Basis-Graphmuster zu einem größeren sternförmigen Graphmuster zusammengeführt werden.

- *Tautologien und Kontradiktionen* – Filterprädikate, die immer zu *wahr* oder *falsch* evaluieren (z. B. `?x = ?x` oder `?x > 1 && ?x < 0`), können durch die jeweiligen Wahrheitswerte ersetzt werden. Im Falle einer Kontradiktion kann diese statische Analyse der Filterausdrücke beispielsweise auch dazu führen, dass es für ein Basis-Graphmuster und damit für alle nachfolgenden Join-Operationen keine Lösungsabbildung geben kann. Diese Teile des SQGM brauchen damit nicht ausgewertet zu werden.

5.3.6 Beispiel einer SQGM-Transformation

Abbildung 5.4 zeigt das resultierende SQGM, nachdem die Transformationsregel 6 (Verschieben von Filteroperatoren) und 7 (Einschränken von Variablenbindungen) auf das SQGM aus Abbildung 5.3 auf Seite 87 angewendet wurde. Die erste Regel führt dazu, dass im transformierten SQGM der Filterausdruck unmittelbar nach dem Graphmuster-Operator ausgeführt wird. Mit der zweiten Regel wurden nicht mehr benötigte Variablenbindungen aus den ausgehenden Iteratoren des Filter-Operators entfernt.

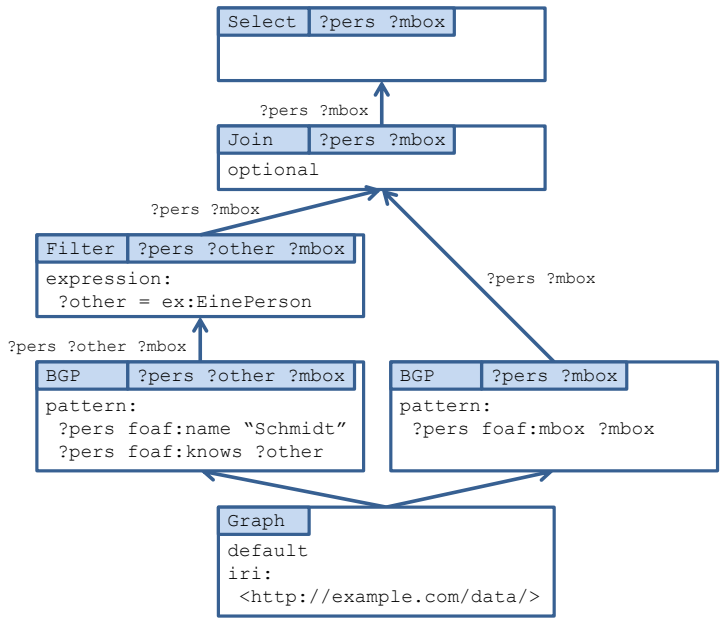


Abbildung 5.4: Resultierendes SQGM nach dem Anwenden von Transformationsregeln auf das SQGM aus Abbildung 5.3

5.4 Ausführungsstrategien für Operatoren

Die zuvor im Abschnitt 5.2 vorgestellten Operatoren und Iteratoren beschreiben eine SPARQL-Anfrage unabhängig von dem RDF-Datenbanksystem. Im Gegensatz dazu werden in diesem Abschnitt Ausführungsstrategien und Zugriffspfade (*engl. access path*) beschrieben, die die Eigenschaften des Speichermodells und der RDF-Daten berücksichtigen. Für einen Operator oder eine Menge von Operatoren wird dabei ein konkreter Algorithmus zur Berechnung der Ausgabedaten beschrieben. Da diese Ausführungsstrategien von der physischen Speicherung der Daten abhängig sind und nicht ohne Weiteres auf andere RDF-Datenbanksystem übertragbar sind, werden diese als *physische* Operatoren bezeichnet.

Den in diesem Abschnitt beschrieben Ausführungsstrategien ist gemein, dass sie die im RCS definierten Subjekt-, Prädikat- und Objektindexe zur Reduktion der Menge der zu betrachtenden Subjekte einsetzen. [HZ11] Somit wird ein Durchlaufen aller Datenbankseiten zur Berechnung des Ergebnisses einer Anfrage vermieden. Im Folgenden werden zunächst grundlegende Strategien zur Evaluation von sternförmigen Graphmustern beschrieben. Diese Strategien werden anschließend dahingehend erweitert, dass gleichzeitig Filterausdrücke berücksichtigt werden. Danach werden Strategien unter Verwendung von Indexen (vgl. Kapitel 4) betrachtet. Am Ende dieses Abschnitts wird auf Algorithmen zur Berechnung von Join-Operatoren eingegangen.

Für jede Ausführungsstrategie wird im Folgenden sowohl ein Algorithmus in Pseudocode als auch eine Abschätzung dessen Komplexität angegeben. Das Maß für die Abschätzung der Komplexität eines Algorithmus' ist dabei die Anzahl der vom Sekundärspeicher zu lesenden Datenseiten.

5.4.1 Sternförmige Graphmuster

Aufgrund des Speichermodells vom Resource Centered Store ist es sinnvoll, eine Anfrage zunächst in größtmögliche sternförmige Graphmuster zu zerlegen und für deren Evaluation spezielle Ausführungsstrategien zu definieren. Diese Strategien nutzen die Eigenschaft aus, dass alle Tripel mit demselben Subjekt auf derselben Datenseite gespeichert sind.

5.4.1.1 Subjektzentrierte Auswertung

Bei der subjektzentrierten Auswertung kann in Betracht gezogen werden, wenn von dem sternförmigen Graphmuster das Subjekt eine Konstante ist. Wie durch den Algorithmus 3.1 auf Seite 35 beschrieben wurde, wird der Subjektindex im RCS ausgenutzt, um die relevante Datenseite zu identifizieren. Die Menge der Lösungsabbildungen für das Graphmuster können unmittelbar durch dessen Abgleich mit den auf der Datenseite gespeicherten Tripeln generiert werden.

Ausführungsstrategie 1.
Anwendungsbereich: $\{\mathsf{op} \in \mathsf{OP} : \mathsf{op}.type = \mathsf{BGP}\}$, wobei $\mathsf{op}.pattern$ sternförmig ist und $\forall t \in \mathsf{op}.pattern \Rightarrow t_s \in \mathbb{I}$ gilt.

Algorithmus: Siehe Algorithmus 3.1 auf Seite 35

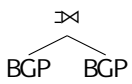
5.4.1.2 Prädikat- und objektzentrierte Auswertung

Der Algorithmus 3.1 beschreibt auch den Fall ab, wenn das Subjekt des Graphmusters eine Variable oder eine anonyme Ressource ist. In diesem Fall werden die Prädikat- und Objektindexe des RCS verwendet, um den Suchraum der Datenseiten weitestgehend einzuschränken. Da bei der Eingrenzung der Datenseite auf Basis dieser Indexe auch Subjekte ohne Relevanz ausgewählt werden können, muss eine Nachprüfung erfolgen.

Ausführungsstrategie 2.
Anwendungsbereich: $\{\mathsf{op} \in \mathsf{OP} : \mathsf{op}.type = \mathsf{BGP}\}$, wobei $\mathsf{op}.pattern$ sternförmig und $\forall t \in \mathsf{op}.pattern \Rightarrow t_s \in \mathbb{V} \cup \mathbb{B}$.
Algorithmus: Siehe Algorithmus 3.1 auf Seite 35

Eine Abschätzung der Komplexität für die beiden zuvor erwähnten Ausführungsstrategien findet sich in Abschnitt 3.3 unter dem Punkt Operationen auf Seite 37.

5.4.1.3 Optionale Graphmuster



Bei der Auswertung eines optionalen Graphmusters P_1 `OPTIONAL` P_2 werden im Allgemeinen die Lösungsabbildungen von P_2 über einen Left-Outer-Join mit denen von P_1 verknüpft. Wenn es sich jedoch bei P_1 um ein Sternmuster handelt und die Tripelmuster in P_2 dasselbe Subjekt wie P_1 haben, dann kann der Left-Outer-Join unmittelbar ohne ein weiteres Laden von Datenseiten ausgeführt werden. Falls es nämlich für P_2 Lösungen gäbe, dann könnten aufgrund des RCS-Speichermodells diese gleichzeitig mit Lösungen zu P_1 berechnet werden. Die Ausführungsstrategie 3 beschreibt den Algorithmus im Detail.

Ausführungsstrategie 3.
Anwendungsbereich: $\{\mathsf{op} \bowtie \in \mathsf{OP} : \mathsf{op}.type = Join\}$, Die eingehenden Iteratoren von $\mathsf{op}_{\bowtie}$ seien $\mathsf{it}_{\mathsf{BGP}} = (\mathsf{op}_{\mathsf{BGP}}, \mathsf{op}_{\bowtie}, v)$ und $\mathsf{it}_{\mathsf{OGP}} = (\mathsf{op}_{\mathsf{OGP}}, \mathsf{op}_{\bowtie}, w)$. Es gelte:

i) $\mathsf{op}.optional = \mathsf{it}_{\mathsf{OGP}}$

ii) $\mathsf{op}_{\mathsf{BGP}}.pattern$ ist sternförmig

iii) $\mathsf{op}_{\mathsf{OGP}}.pattern$ hat dasselbe Subjekt wie $\mathsf{op}_{\mathsf{BGP}}.pattern$

Algorithmus: Algorithmus 5.1 ist eine Erweiterung des Algorithmus 3.1 (Seite 35). Zeile 2 umschreibt vereinfacht den Vorgang, dass mittels der Subjekt-, Prädikat- und Objektindexe die Adressen der relevanten Subjekte bestimmt werden (vgl. Algorithmus 3.1). Für jede dieser ermittelten Adressen wird die entsprechende Datenseite geladen und zunächst die Lösungsabbildungen für $\mathsf{op}_{\mathsf{BGP}}$ berechnet (Zeilen 5 und 6). Für das adressierte Subjekt werden danach die Lösungsabbildungen für $\mathsf{op}_{\mathsf{OGP}}$ berechnet (Zeile 7). Abschließend werden die beiden Multimengen von Lösungsabbildungen miteinander verknüpft (Zeile 8).

In dem zuvor genannten Algorithmus hat die Evaluation des optionalen Graphmusters $\mathsf{op}_{\mathsf{OGP}}.pattern$ keinen Einfluss auf die Komplexität, da der

Algorithm 5.1 matchOptionalStarBGP(op_{BGP}, op_{OGP}) : Ω

```
1:  bindings = ∅
2:  Ermittle bitset der relevanten Subjekte mittels Indexe
3:  addresses = subjectIdx.get(bitset)      // Seiten-IDs der Subjekte ermitteln
4:  for address : addresses do
5:      page = load(address.pageId)
6:      bindingsBGP = match(page, address.slot, op_BGP.pattern)
7:      bindingsOGP = match(page, address.slot, op_OGP.pattern)
8:      bindings += bindingsBGP ⟕ bindingsOGP
9:  end for
10: return bindings
```

Left-Outer-Join während der Auswertung von $op_{BGP}.pattern$ berechnet werden kann. Damit ergibt sich dieselbe Komplexität wie im Algorithmus 3.1 (vgl. Abschnitt 3.3, Unterpunkt Operationen auf Seite 37).

Beispiel 5.2. Abbildung 5.5 illustriert das Prinzip dieser Ausführungsstrategie. Obwohl im Anfragemodell einen Join-Operator beinhaltet, kann aufgrund der Ausführungsstrategie 3 ein Ausführungsplan ohne die Verwendung eines Join-Algorithmus' erstellt werden. Da durch die Operatoren im grau hinterlegten Bereich ein sternförmiges Graphmuster ausgewertet wird, kann aufgrund des RCS-Speichermodells der optionale Teil zeitgleich mit anderen Graphmuster ausgewertet werden. □

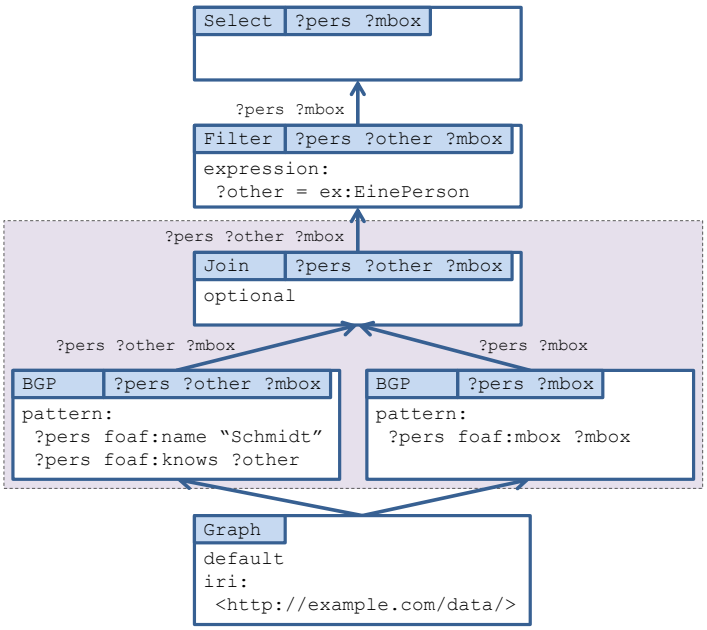


Abbildung 5.5: Ausführungsstrategie für ein sternförmiges Graphmuster und einem optionale Tripelmuster

5.4.2 Auswertung von Filterausdrücken

FS
|
BGP

Ein grundsätzlicher Ansatz zur Reduktion des Datenvolumens bei der Verwaltung von RDF-Daten ist das Substituieren von IRIs und Literalen durch nummerische Werte oder Hashwerte (vgl. Abschnitt 3.1.1 auf Seite 22). Die Verwendung einer solchen ID birgt jedoch den Nachteil, dass bei der Evaluation von Filterausdrücken nur Prädikate der Form `?x = k` ohne Weiteres unterstützt werden.

Wenn Prädikate eines Filterausdrucks einen Wertebereich abtesten oder Zeichenketten in IRIs oder Literalen gesucht werden, sind zusätzliche Schritte erforderlich. Abgesehen davon, dass die Abbildung zwischen IRI bzw. Literal und ID bei Textsuchen berücksichtigt werden muss, ist die Vergabe der IDs im Allgemeinen nicht ordnungserhaltend. Ein naïver Ansatz zum Auswerten von derartigen Filterausdrücken wäre deren Evaluation, nachdem alle Lösungsabbildungen berechnet worden sind. Bei hoher Selektivität des Filterausdrucks ist ein solches Vorgehen nicht vertretbar.

Daher werden in diesem Abschnitt RCS-spezifische Ausführungsstrategien beschrieben, die ein möglichst frühzeitiges auswerten der Filterausdrücke ermöglichen. Da die Auswertung von Anfragen im RCS auf sternförmigen Graphmustern basiert, bilden die im vorherigen Abschnitt genannten Algorithmen die Basis.

5.4.2.1 Prädikate der Form `?x = k`

Die Subjekt-, Prädikat- und Objektindexe des Resource Centered Store können nicht nur zum Auswerten eines Basis-Graphmuster-Operators verwendet werden. Sie können auch dazu herangezogen werden, um gleichzeitig zur Auswertung eines sternförmigen Basis-Graphmusters bestimmte Prädikate eines Filter-Operators zu evaluieren. Die Voraussetzungen für diese Ausführungsstrategie sind, dass zum einen das Graphmuster $\mathsf{op}.pattern$ des Basis-Graphmuster-Operator op sternförmig und dessen Subjekt eine Variable $?s$ ist. Zum anderen ist op Datenproduzent für einen Filter-Operator $\mathsf{op}_{\mathsf{FS}}$, dessen Filterausdruck $\mathsf{op}_{\mathsf{FS}}.expression$ in eine disjunktive Normalform von Prädikaten der Form $?x = k$ mit $k \in \mathbb{I} \cup \mathbb{L}$ überführt werden kann. Der Filterausdruck $\mathsf{op}_{\mathsf{FS}}.expression$ hat dementsprechend die folgende Form:

$$\underbrace{(?x_{11} = k_{11} \wedge \ldots \wedge ?x_{1n} = k_{1n})}_{e_1} \vee \ldots \vee \underbrace{(?x_{m1} = k_{m1} \wedge \ldots \wedge ?x_{ml} = k_{ml})}_{e_m}$$

In Abhängigkeit vom Auftreten der Variablen $?s$ im Filterausdruck können zwei unterschiedliche Ausführungsstrategien angewendet werden. Bei der ersten Strategie 4 kommt die Bedingungen hinzu, dass die Variable s in jedem konjunktiven Teil $e_1 \ldots e_m$ genau einmal in einem Prädikat auftritt. Falls derartige Prädikate mehrfach mit unterschiedlichen Konstanten k_i und k_j in demselben konjunktiven Teil auftreten, dann kann dieser Teil komplett ignoriert werden – er evaluiert immer zu *falsch*.

Ausführungsstrategie 4.
Anwendungsbereich: $\{\mathsf{op} \in \mathsf{OP} : \mathsf{op}.type = \mathsf{BGP}\}$, wobei Folgendes gilt:

i) op.*pattern* ist sternförmig mit Variable ?s als Subjekt

ii) $\exists \mathsf{it} \in \mathsf{IT} : \mathsf{it} = (\mathsf{op}, \mathsf{op}_{\mathsf{FS}}, v) \wedge v \subseteq \mathrm{var}(\mathsf{op})\}$

iii) $\mathsf{op}_{\mathsf{FS}}$.*expression* ist disjunktive Normalform von Prädikaten der Form ?$x = k$ mit $k \in \mathbb{I} \cup \mathbb{L}$

iv) $e_1 \ldots e_m$ enthalten genau ein Prädikat der Form ?$s = k_i$

Algorithmus: Die Ausführungsstrategie basiert auf dem Algorithmus 3.1 für den Fall, wenn das Subjekt des auszuwertenden Graphmusters eine Konstante ist. Im Algorithmus 5.2 wird analog für jede normalisierte Konstante $\iota(k_i)$[4] mit Hilfe des Subjekt-Indexes die ID der Datenseite ermittelt, die alle Tripel mit dem Subjekt k_i enthält (Zeile 3). Diese Datenseiten werden anschließend geladen, die Lösungsabbildungen für das Graphmuster op.*pattern* bestimmt (Zeilen 4 und 5). Für die gewonnenen Lösungsabbildungen wird jeweils der Filterausdruck $\mathsf{op}_{\mathsf{FS}}$.*expression* ausgewertet (Zeile 6).

Algorithm 5.2 matchFilteredStarBGP($\mathsf{op}_b gp$, $\mathsf{op}_f s$) : Ω

```
1: bindings = ∅
2: for ?s = k_i ∈ e_1 ... e_m do
3:    address = subjectIdx.get(ι(k_i))
4:    page = load(address.pageId)
5:    bindings = match(page, address.slot, op_BGP.pattern)
6:    Entferne Lösungsabb. aus bindings, die op_FS.expression nicht erfüllen
7: end for
8: return bindings
```

Die Komplexität von Algorithmus 5.2 ergibt sich aus der Anzahl n der konjunktiven Teile im Filterausdruck $\mathsf{op}_{\mathsf{FS}}$.*expression*, da jedes genau eine Ressource als Subjekt festlegt. Für jedes dieser Subjekte wird über den Subjektindex (B+-Baum) die ID der zu ladenden Datenseite ermittelt und anschließend vom Sekundärspeicher geladen. Damit ergibt sich im schlimmsten Fall eine Komplexität von $O(n \log_b M)$.

In der zweiten Strategie wird davon ausgegangen, dass die Variable s in keinem Prädikat des Filterausdrucks auftritt.

Ausführungsstrategie 5.
Anwendungsbereich: $\{\mathsf{op} \in \mathsf{OP} : \mathsf{op}.type = \mathsf{BGP}\}$, wobei Folgendes gilt:

i) op.*pattern* ist sternförmig mit Variable ?s als Subjekt

ii) $\exists \mathsf{it} \in \mathsf{IT} : \mathsf{it} = (\mathsf{op}, \mathsf{op}_{\mathsf{FS}}, v) \wedge v \subseteq \mathrm{var}(\mathsf{op})\}$

iii) $\mathsf{op}_{\mathsf{FS}}$.*expression* ist disjunktive Normalform von Prädikaten der Form ?$x = k$ mit $k \in \mathbb{I} \cup \mathbb{L}$.

iv) $e_1 \ldots e_m$ enthält kein Prädikat der Form ?$s = k_i$

[4] vgl. Normalisierungsfunktion ι in Definition 3.3 auf Seite 30

Algorithmus: Algorithmus 5.3 ergänzt die Berechnung des Bitsets in Algorithmus 3.1, Zeilen 8 bis 22. Im Algorithmus 5.3 wird ein Bitset *bitset* generiert, dass die Datenseiten und Subjekte repräsentiert, die Lösungsabbildungen für das Graphmuster unter Berücksichtigung des Filterausdrucks beitragen können. Dieses Bitset wird anschließend mit dem Bitset aus Algorithmus 3.1 verundet, so dass die konstanten Werte aus dem Graphmuster auch berücksichtigt werden.

Im Detail funktioniert der Algorithmus wie folgt: Zunächst werden zwei leere Bitsets erzeugt. Das eine wird im Fall von Filterprädikaten ($?x = k_i$) verwendet, bei denen die Variable als Prädikat im Graphmuster vorkommt, und das andere, wenn sie als Objekt vorkommt. Nacheinander werden die konjunktiven Teile $e_1 \ldots e_m$ des Filterausdruck FS.*expression* betrachtet.

Für jedes Prädikat wird bestimmt, ob die Variable als Prädikat und/oder als Objekt im Graphmuster vorkommt und dementsprechend das Bitset für die Konstante k_i aus dem Prädikat- bzw. Objektindex geladen. Diese wird mit einem temporären Prädikat- und/oder Objektbitset verundet (Zeilen 7 bis 12). Nachdem alle alle Prädikate eines konjunktiven Teils betrachtet worden sind, werden die temporären Bitsets mit dem jeweiligen Prädikat- und Objektbitset verodert (Zeilen 14 und 15). Das Ergebnis ergibt sich aus der Verundung des Prädikat- und Objektbitsets.

Algorithm 5.3 createEquiFilterBitsets(op_{FS}, op_{BGP}) : Bitset

```
1:  predicateBitset = 0                                  // Bitset ohne gesetzte Bits
2:  objectBitset = 0
3:  for e_i ∈ op_FS.expression do
4:     tmpPredicateBitset = ~ 0                          // Bitset mit allen Bits gesetzt
5:     tmpObjectBitset = ~ 0
6:     for ?x_ij = k_ij in e_i do
7:        if isPredicate(?x, op_BGP.pattern) then
8:           tmpPredicateBitset &= predicateIdx.get(ι(k_ij))
9:        end if
10:       if isObject(?x, op_BGP.pattern) then
11:          tmpObjectBitset &= objectIdx.get(ι(k_ij))
12:       end if
13:    end for
14:    predicateBitset |= tmpPredicateBitset
15:    objectBitset |= tmpObjectBitset
16: end for
17: return  predicateBitset & objectBitset
```

Die Komplexität von Algorithmus 5.3 wird im Wesentlichen durch den Zugriff auf den Prädikat- und Objektindex für das Ermitteln der Bitsets für die jeweiligen Konstanten der Filterprädikate bestimmt. Wenn n die Anzahl der Filterprädikate im Ausdruck $op_{FS}.expression$ ist, dann hat der Algorithmus im schlimmsten Fall – jede Konstante tritt gleichzeitig als Prädikat und als Objekt auf – die folgende Komplexität:

$$O(\underbrace{2n\log_b M}_{\text{B+-Baum}} + \underbrace{2n \cdot 2N}_{\text{Bitsets}}) = O(2n(\log_b M + 2N))$$

wobei $\log_b M$ der Komplexität des Zugriffs auf die B+-Bäume des Prädikat- und Objektindexes und $2N$ das Laden der Bitsets darstellt. An dieser Stellen sei noch einmal darauf hingewiesen, dass für das Laden eines Bitsets im Allgemeinen nur auf eine Datenseite zugegriffen werden muss (vgl. Abschnitt 3.3, Unterpunkt Indexe). Neben dem in der obigen Gleichung angegebenen Aufwand müssen noch die Kosten für das Evaluieren des Graphmusters hinzugerechnet werden (vgl. Abschnitt 3.3, Unterpunkt Operationen).

5.4.2.2 Bereichsanfragen und Textsuche

Während sich Bereichsanfragen auf Filterausdrücke mit Prädikaten der Form `?x` θ `k` mit $\theta \in \{<, >, <=, >=\}$ beziehen, geht es bei der Textsuche um das Finden von einer Zeichenkette oder einem regulären Ausdruck in einer anderen Zeichenkette (z. B. Zeichenkettenfunktionen `contains()`, `strstarts()` oder `regex()`).

Im Allgemeinen können diese Filterausdrücke und insbesondere Prädikate, die sich nur auf einen Teil einer IRI oder eines Literals beziehen, nicht über die durch die Normalisierung zugewiesenen IDs ausgewertet werden. Bereichsanfragen könnten nur dann über den IDs ausgewerteten werden, falls als Normalisierungfunktion für IRIs und Literale eine ordnungserhaltene Hashfunktion eingesetzt wird.

Daher wird für das Verwenden der folgende Ausführungsstrategie eine Normalisierungsfunktion bzw. eine Indexstruktur für die Verwaltung der Normalisierungsabbildung vorausgesetzt, die ein effizientes Auswerten von Bereichsanfragen und Zeichenkettenfunktionen erlaubt. Im Folgenden wird beides unter dem Begriff Normalisierungsindex zusammengefasst.

Da die Ausführungsstrategie eine Erweiterung der zuvor beschriebenen Algorithmen 5.2 und 5.3 ist, werden im Folgenden nur Prädikate[5] betrachtet, die einen Bereich spezifizieren oder eine Zeichenkettenfunktion beinhalten.

Ausführungsstrategie 6.
Anwendungsbereich: $\{\mathsf{op} \in \mathsf{OP} : \mathsf{op}.type = \mathsf{BGP}\}$, wobei Folgendes gilt:

i) $\mathsf{op}.pattern$ ist sternförmig mit Variable $?s$ als Subjekt

ii) $\exists \mathsf{it} \in \mathsf{IT} : \mathsf{it} = (\mathsf{op}, \mathsf{op}_{\mathsf{FS}}, v) \wedge v \subseteq \mathrm{var}(\mathsf{op})\}$

iii) $\mathsf{op}_{\mathsf{FS}}.expression$ ist disjunktive Normalform von Prädikaten der Form $?x = k$ mit $k \in \mathbb{I} \cup \mathbb{L}$.

iv) $e_1 \ldots e_m$ enthält ein Prädikat der Form $?s\theta k_i$ oder eine Zeichenkettenfunktion

Algorithmus: Zunächst sei die Variable des Filterprädikats $?x$ das Subjekt des Graphmusters. Dann werden über den Normalisierungsindex alle IDs von IRIs

[5] Die Ausführungsstrategie lässt sich sowohl auf ein- als auch beidseitig begrenzte Bereichsanfragen anwenden.

bestimmt, die das Filterprädikat erfüllen. Literale müssen bei der Auswertung des Filterprädikats nicht berücksichtigt werden. Die Zeilen 3 bis 6 von Algorithmus 5.2 werden für jede ermittelte ID ausgeführt.

Alternativ kann die Variable als Prädikat bzw. Objekt im Graphmuster vorkommen. Wie zuvor werden die das Filterprädikat erfüllenden IDs mit Hilfe des Normalisierungsindex bestimmt. Nur falls die Variable als Objekt im Graphmuster verwendet wird, muss das Filterprädikat auch für Literale überprüft werden. Für diese IDs werden anschließend aus dem Prädikat- bzw. Objektindex die entsprechenden Bitsets ermittelt und mit dem Bitset *tmpPredicateBitset* bzw. *tmpObjectBitset* verundet. Das heißt, die Zeilen 7 bis 12 des Algorithmus' 5.3 werden für jede ID ausgeführt.

Die Komplexität des Algorithmus setzt sich aus Aufwand für das Ermitteln der IDs aus dem Normalisierungsindex und dem Zugriff auf den jeweiligen Index zusammen. Der erste Teil besteht aus dem Lokalisieren der Konstanten im Index, $O(\log_b M)$, und dem sequentiellen Lesen des Wertebereichs, $O(n)$ mit n = Anzahl der Werte in dem betrachteten Bereich. Die Komplexität des zweiten Teils hängt davon ab, ob Subjekt-Index oder Prädikat-/Objektindex zugegriffen wird. Der Zugriff auf den Subjekt-Index hat die Komplexität $O(n \log_b M)$ und die Zugriffe auf die Prädikat-/Objektindex haben wie in der zuvor besprochenen Ausführungsstrategie die Komplexität $O(2n \log_b M + 2n \cdot 2N)$. Die Anzahl der zu betrachtenden IDs ist somit für die Komplexität der Ausführungsstrategie maßgeblich.

5.4.3 Indexbasierte Strategien

In diesem Abschnitt werden Strategien beschrieben, die entweder den Zugriff auf den zu einem Graphmuster gehörenden Index oder auf die Indexe des RCS beinhalten. Zu Beginn der jeweiligen Abschnitte wird auf der rechten Seite der betrachtete Teil einer Anfrage symbolisch dargestellt.

5.4.3.1 Indexierte Graphmuster

Die Voraussetzungen für die Verwendung der nachfolgend beschriebenen Ausführungsstrategien sind zum einen, dass eine Menge von Indexen über dem RDF-Graphen definiert worden sind, und zum anderen, dass mindestens einer dieser Indexe für das gegebene Graphmuster zulässig ist (vgl. Definition 4.3.1 auf Seite 62). Dabei musste das Indexmuster nicht exakt mit dem Graphmuster übereinstimmen, sondern es genügte die Existenz einer Graphmuster-Abbildung. Zum Beispiel konnte das Graphmuster anstelle einer Variablen auch einen konkreten Wert haben. Da jeder Index als Tabelle in einem relationalen DMBS gespeichert wird, entspricht der Zugriff auf einen Index grundsätzlich dem Scan aller Einträge in dieser Tabelle. Diese werden dann an den konsumierenden Operator weitergegeben.

I_{BGP}

Ausführungsstrategie 7.
Anwendungsbereich: $\{\mathsf{op} \in \mathsf{OP} : \mathsf{op}.type = BGP\}$. Es gelte:

i) $\exists I = (P, \Omega_P) \in \mathcal{I} : I$ ist zulässig für $\mathsf{op}.pattern$

ii) ν die Variablenabbildung von P auf $\mathsf{op}.pattern$

Algorithmus: Zuerst wird anhand der Variablenabbildung ν ein Filterausdruck erzeugt, um später die nicht zur Lösung von op.*pattern* gehörenden Lösungsabbildungen entfernen zu können. Anschließend werden die Lösungsabbildungen aus dem Index geladen und gefiltert zurückgegeben.

Algorithm 5.4 indexAccess(op, $I = (P, \Omega_P)$) : Variablenbindungen

1: Bestimme ν für op.*pattern* und P
2: Erzeuge aus ν einen Filterausdruck *expression*
3: bindings = Ω_P
4: Entferne Lösungsabb. aus bindings, die *expression* nicht erfüllen
5: **return** bindings

Der Zeitpunkt der Auswertung des Filterausdrucks ist von dem Speichersystem abhängig, das für die Verwaltung der Indexe verwendet wird. Da beispielsweise im RCS für die Speicherung der Indexe eine relationale Datenbank eingesetzt wird, kann der Filterausdruck als Teil der `WHERE`-Klausel direkt vom RDBMS evaluiert werden.

Der Aufwand für die Berechnung der Lösungsabbildungen für ein indexiertes Graphmuster hängt ausschließlich mit der Anzahl der Einträge im Index zusammen und ist damit unabhängig von der Größe des Graphmusters. Die Komplexität ist somit $O(|I|)$.

5.4.3.2 Ausschließlicher Indexzugriff

ASK
|
I_{BGP}

Für einige Anfragen kann die Lösung ohne jeglichen Zugriff auf Datenseiten bestimmt werden. Im Folgenden werden derartige Anfragen als *indexlösbar* bezeichnet. Wenn eine `ASK`-Anfrage aus sternförmigen Basis-Graphmustern mit einer bestimmten Form bestehen, kann ausschließlich anhand des Subjekt-, Prädikat- und Objektindex die Existenz von Lösungsabbildungen nachgewiesen werden. Zunächst wird in der nachfolgenden Ausführungsstrategie der Fall betrachtet, dass die `ASK`-Anfrage aus einem einzelnen Basis-Graphmuster besteht. Anschließend wird ein Kriterium angegeben, unter welchem eine Anfrage bestehend aus mehreren Basis-Graphmustern indexlösbar ist.

Falls die Anfrage aus einem einzelnen Basis-Graphmuster besteht, dann ist sie indexlösbar, wenn das Graphmuster sternförmig ist und in jedem Tripelmuster das Prädikat oder das Objekt eine Variable ist. Darüber hinaus wird gefordert, dass in zwei Tripelmuster zwei verschiedene Variablen in der Prädikatposition verwendet werden müssen, wenn diese Objekt nicht übereinstimmen. Eine analoge Bedingung wird für Variablen in Objektpositionen formuliert. Diese Bedingungen sind erforderlich, da über das Kombinieren der Bitsets aus dem Prädikat- und Objektindexen nicht bestimmt werden kann, ob ein Tripel mit einem bestimmten Prädikat und einem bestimmten Objekt für ein Subjekt existiert.

Die nachfolgend beschriebene Ausführungsstrategie fasst die Bedingungen und den Algorithmus zusammen.

Ausführungsstrategie 8.
Anwendungsbereich: Anfragepläne $Q = (\mathsf{OP}, \mathsf{IT}, r, dflt, NG)$ mit folgenden Eigenschaften:

i) $\mathsf{OP} = \{\mathsf{op}_A, \mathsf{op}_{\mathsf{BGP}}\} \wedge \mathsf{IT} = \{\mathsf{it}\} \wedge r = \mathsf{op}_A$

ii) $\mathsf{op}_A.type = \text{Ask}$

iii) $\mathsf{op}_{\mathsf{BGP}}.type = \text{BGP} \wedge \mathsf{op}_{\mathsf{BGP}}.pattern$ ist sternförmig $\wedge \forall t \in \mathsf{op}_{\mathsf{BGP}}.pattern : t^p \in \mathbb{V} \vee t^o \in \mathbb{V}$

iv) $\forall t_1, t_2 \in \mathsf{op}_{\mathsf{BGP}}.pattern : t_1^p \neq t_2^p \Rightarrow t_1^p \neq t_2^p$

v) $\mathsf{it} = (\mathsf{op}_{\mathsf{BGP}}, \mathsf{op}_A, \emptyset)$

Algorithmus: Wie bei einigen zuvor beschriebenen Ausführungsstrategien werden für jede Konstante im Graphmuster die entsprechenden Bitsets aus dem Prädikat- bzw. Objektindex bestimmt, die anschließend miteinander verundet werden (Zeilen 2 bis 9. Im Ergebnis erhält man ein Bitset, dass das die Adressen derjenigen Subjekte angibt, die für das Beantworten der Anfrage relevant sind. Falls das Subjekt des Graphmusters eine Variable ist, dann hat die ASK-Anfrage genau dann die Lösung *wahr*, wenn das Bitset mindestens ein gesetztes Bit beinhaltet (Zeile 14). Falls hingegen das Subjekt eine Ressource ist, muss zusätzlich überprüft werden, ob das richtige Bit für diese Ressource im Bitset gesetzt ist (Zeile 12).

Algorithm 5.5 indexAsk($\mathsf{op}_{\mathsf{BGP}}$) : {*wahr*, *falsch*}

```
1:  bitset = ~ 0                                   // Bitset mit allen Bits gesetzt
2:  for t ∈ op_BGP.pattern do
3:     if (isResource(t.preidicate)) then
4:        bitset &= predicateIdx.get(t.predicate)
5:     end if
6:     if (isResource(t.object) || isLiteral(t.object)) then
7:        bitset &= objectIdx.get(t.object)
8:     end if
9:  end for
10: if isResource(t.subject) then
11:    address = subjectIdx.get(t.subject)
12:    return wahr, falls Bit address in bitset gesetzt; falsch, sonst
13: else
14:    return wahr, falls ein Bit in bitset gesetzt; falsch, sonst
15: end if
```

Abbildung 5.6 illustriert die Notwendigkeit der Bedingungen iii) und iv. Der Prädikatindex beinhaltet für `foaf:name` und `foaf:givenName` jeweils ein Bitset, in dem das Bit für die Adresse von `ex:pers` gesetzt ist. Die Objekte für diese Prädikate stimmen jedoch nicht überein.

Es sei noch angemerkt, dass eine zu der oben beschriebenen korrespondierende SELECT-Anfrage nicht beantwortet werden kann – selbst dann nicht, wenn das Schlüsselwort DISTINCT angegeben werden würde. Die Ursache dafür ist in der Konstruktion des Sujektindexes zu finden. Über das berechnete Bitset können zwar die Adressen der Subjekte ermittelt werden können, aber der Subjektindex erlaubt keinen effizienten Zugriff auf die Ressource-ID bei gegebener Adresse.

```
ex:pers foaf:name "Hans␣Meyer".
ex:pers foaf:givenName "Hans".
```

```
ASK WHERE {
  ?person foaf:name ?name.
  ?person foaf:givenName ?name.
}
```

Abbildung 5.6: Anfrage ohne Lösung, aber Algorithmus 5.5 liefert ein Bitset mit gesetztem Bit für `ex:pers`

Wenn die `ASK`-Anfrage aus mehreren sternförmigen Graphmustern besteht, dann ist die Anfrage genau dann lösbar, wenn zum einen jedes einzelne sternförmige Graphmuster die Bedingungen iii) und iv) erfüllt und zum anderen diese *nicht* über eine gemeinsame Variable verfügen. Die zweite Bedingung ist notwendig, da mittels der Indexe kein Join zwischen den Graphmustern berechnet werden kann. Falls die Graphmuster jedoch über keine gemeinsame Variable verfügen, dann wird die Lösung des gesamten Graphmuster durch ein Kreuzprodukt berechnet. Dieses Kreuzprodukt hat genau dann mindestens eine Lösung, wenn jedes sternförmige Graphmuster mindestens eine Lösung hat. Das heißt, wenn die korrespondierenden Bitsets mindestens ein gesetztes Bit beinhalten.

Abbildung 5.7 zeigt ein Beispiel, bei dem die zwei sternförmigen Graphmuster (jeweils ein Tripelmuster) über eine Variable miteinander verbunden sind. Der Algorithmus 5.5 würde in diesem Fall *wahr* zurückgegeben, obwohl die das Graphmuster nicht in der Datenbasis enthalten ist.

```
ex:pers1 foaf:name "Hans␣Meyer".
ex:pers2 foaf:name "Knut␣Meyer".
```

```
ASK WHERE {
  ?pers1 foaf:name ?name.
  ?pers2 foaf:name ?name.
}
```

Abbildung 5.7: Anfrage ohne Lösung, aber Algorithmus 5.5 liefert ein Bitset mit gesetztem Bit für `ex:pers1` und `ex:pers2`

Die Komplexität von Algorithmus 5.6 bestimmt sich über die Anzahl k der Tripelmuster im Graphmuster. Für jedes Tripelmuster muss entweder der Prädikat- oder der Objektindex zugegriffen werden. Daraus ergibt sich eine Komplexität von $O(k(\log_b M + 2N))$, der erste Teil der Summe repräsentiert einen B+-Baum-Zugriff und der zweite das Laden des Bitsets. Im schlimmsten Szenario, bei dem in allen n sternförmigen Graphmustern das Subjekt eine Ressource ist, ändert sich die Komplexität zu $O((k+n)(\log_b M + 2N))$.

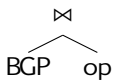
Diese Strategie kann auch dann eingesetzt werden, wenn nur die Existenz eines Graphmusters überprüft werden muss, d.h. die Anfrage muss nicht unbedingt vom Typ `ASK` sein. Mit Hilfe dieses Operators könnte beispielsweise die `NOT EXIST`-Klausel von SPARQL Version 1.1 unterstützt werden.

5.4.4 Join-Strategien mit Basis-Graphmuster-Operator

Aufgrund der Eigenschaften des RCS-Speichermodells wird eine Evaluation von sternförmigen Graphmuster begünstigt. Daher werden in einem Ausfüh-

rungsplan für die Auswertung einer SPARQL-Anfrage häufig Join-Operationen unter Beteiligung eines Basis-Graphmuster-Operators mit einem sternförmigen Graphmuster auftreten. In diesem Abschnitt werden daher speziell Ausführungsstrategien einen derartigen Join betrachtet.

Im Folgenden seien also ein Join-Operator $op_{\bowtie}$, ein Basis-Graphmuster-Operatoren op_{BGP} mit dem sternförmigen Graphmuster $P = op_{BGP}.pattern$ sowie ein weiterer Operator op mit bekannter Lösungsabbildung Ω gegeben. Wenn nun eine oder mehr Variablen aus var(op) im Graphmuster P auftreten, können die Lösungsabbildungen in Ω zum Einschränken der relevanten Datenseiten für die Berechnung von op_{BGP} verwendet werden.



5.4.4.1 Subjektindex-basierter Join

Falls (genau) eine Variable ?s des Operators op das Subjekt von P ist, sind die Variablenbindungen für ?s potentiell Teil der Lösungsabbildungen von op_{BGP}. Diese Variablenbindungen können also dazu genutzt werden, um über den Subjektindex relevante Datenseiten für die Berechnung von op_{BGP} zu bestimmen.

Ausführungsstrategie 9.
Anwendungsbereich: $\{op_{\bowtie} \in OP : op_{\bowtie}.type = Join\}$. Die eingehenden Iteratoren von $op_{\bowtie}$ seien $it = (op, op_{\bowtie}, v)$ und $it_{BGP} = (op_{BGP}, op_{\bowtie}, w)$. Es gelte:

i) $op_{BGP}.pattern$ ist sternförmig

ii) $\exists s \in v$: ?s ist Subjekt von $op_{BGP}.pattern$

Algorithmus: Die Berechnung der Lösungsabbildungen für den Join-Operator basiert auf Algorithmus 5.2. Für jede Variablenbindung von ?$s \in \Omega$ wird über den Subjektindex die Datenseite bestimmt und dann geladen (Zeilen 2 bis 4). Anschließend werden die Lösungsabbildungen für das Graphmuster P ermittelt (Zeile 5) und mit den Lösungsabbildungen in Ω vereinigt[6] (Zeile 7).

Algorithm 5.6 subjectIndexJoin(op_{BGP}, Ω) : Lösungsabbildungen

```
1:  bindings = ∅
2:  for id ∈ ⋃_{μ∈Ω} μ(?s) do
3:      address = subjectIdx.get(id)
4:      page = load(address.pageId)
5:      tmpBindings = match(page, address.slot, op_BGP.pattern)
6:      for μ' ∈ tmpBindings do
7:          Ergänze bindings um {μ' ∪ μ : μ ∈ Ω}
8:      end for
9:  end for
10: return bindings
```

Die Komplexität des Algorithmus wird im Wesentlichen über die Anzahl der Variablenbindungen für ?s bestimmt. Für jede Variablenbindung muss

[6] vgl. Definition 2.10 auf Seite 12

im schlimmsten Fall einmal der Subjektindex zugegriffen und eine Datenseite geladen werden. Damit ergibt sich für diesen Fall eine Komplexität von $O(k \log_b M)$, wobei k die Anzahl der Variablenbindungen bezeichnet.

5.4.4.2 Prädikat-/Objektindex-basierter Join

Im Folgenden wird der Fall betrachtet, dass die Ergebnisse zweier Operatoren miteinander verknüpft werden sollen, wobei der erste ein Basis-Graphmuster-Operators $\mathsf{op}_{\mathsf{BGP}}$ und er andere ein beliebiger anderer Operator mit bekanntem Zwischenergebnis Ω. Der Prädikat- bzw. Objektindex können dann zur Berechnung eines Operators $\mathsf{op}_{\mathsf{BGP}}$ herangezogen werden, wenn die Menge Ω Bindungen für Variablen beinhaltet, die als Prädikat und/oder Objekt im Graphmuster $\mathsf{op}_{\mathsf{BGP}}.pattern$ vorkommen. Für diese Variablen werden die Bitsets aus dem Prädikat- und Objektindex geeignet miteinander verknüpft, um die Menge der relevanten Datenseiten im Algorithmus 3.1 weiter einzuschränken. Hierfür müssen aus der Multimenge Ω nur die distinkten Lösungsabbildungen betrachtet werden.

Ausführungsstrategie 10.
Anwendungsbereich: $\{\mathsf{op}_{\bowtie} \in \mathsf{OP} : \mathsf{op}_{\bowtie}.type = Join\}$. Die eingehenden Iteratoren von $\mathsf{op}_{\bowtie}$ seien $\mathsf{it} = (\mathsf{op}, \mathsf{op}_{\bowtie}, v)$ und $\mathsf{it}_{\mathsf{BGP}} = (\mathsf{op}_{\mathsf{BGP}}, \mathsf{op}_{\bowtie}, w)$. Es gelte:

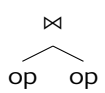
i) $\mathsf{op}_{\mathsf{BGP}}.pattern$ ist sternförmig

ii) $\exists ?s \in v :$?s ist Prädikat und/oder Objekt in $\mathsf{op}_{\mathsf{BGP}}.pattern$

Algorithmus: Für jede Lösungsabbildung werden für die in $\mathsf{op}_{\mathsf{BGP}}.pattern$ als Prädikat verwendeten Variablen die Bitsets über den Prädikatindex ermittelt. Diese werden anschließend miteinander verundet. Die so für jede Lösungsabbildung bestimmten Bitsets werden dann miteinander verodert. Analog werden für die als Objekt vorkommenden Variablen die Bitsets ermittelt, verundet und verodert. Im Algorithmus 5.7 sind dies die Zeilen 7 bis 12 bzw. 14 bis 15. Die beiden so erhaltenen Bitsets werden abschließend miteinander verundet. Als Ergebnis dieser Berechnungen erhält man ein Bitset, das die Adressen derjenigen Subjekte kodiert, die zu Lösungsabbildungen für $\mathsf{op}_{\mathsf{BGP}}$ führen können.

Die Komplexität dieses Algorithmus ist von der Anzahl der Subjekte und der Selektivität von $\mu[\mathsf{BGP}]$ abhängig. Für jede Variablenbindung von $?s$ muss sowohl der Prädikat- als auch der Objektindex zugegriffen werden. Somit ergibt sich für diesen Algorithmus eine Komplexität von $O(2k \log_b M)$, wobei k die Anzahl der Variablenbindungen bezeichnet.

5.4.5 Allgemeine Join-Strategien



Aufgrund des RCS-Speichermodells besteht die grundlegende Ausführungsstrategie darin, eine SPARQL-Anfrage in Sternmuster maximaler Größe zu zerlegen, die Ergebnisse für diese zu berechnen und diese an andere Operatoren weiterzuleiten. Mit den zuvor beschriebenen Ausführungsstrategien hat der Optimierer nun auch die Möglichkeit, mehrere Operatoren gleichzeitig auswerten zu lassen. Fast alle Ausführungsstrategien haben jedoch gemein,

Algorithm 5.7 predicateObjectJoin(op_{BGP}, Ω) : Bitset

```
1:  predicateBitset = 0                                  // Bitset ohne gesetzte Bits
2:  objectBitset = 0
3:  for μ ∈ Ω do
4:     tmpPredicateBitset = ∼ 0                          // Bitset mit allen Bits gesetzt
5:     tmpObjectBitset = ∼ 0
6:     for ?x ∈ var(op_BGP.pattern do
7:        if isPredicate(?x, op_BGP.pattern) then
8:           tmpPredicateBitset &= predicateIdx.get(μ(?x))
9:        end if
10:       if isObject(?x, op_BGP.pattern) then
11:          tmpObjectBitset &= objectIdx.get(μ(?x))
12:       end if
13:    end for
14:    predicateBitset |= tmpPredicateBitset
15:    objectBitset |= tmpObjectBitset
16: end for
17: return predicateBitset & objectBitset
```

dass sie eine Multimenge von Lösungsabbildungen zum Ergebnis haben. Um am Ende der Anfrageausführung eine einzige Multimenge von Lösungsabbildungen zu erhalten, müssen diese Zwischenergebnisse miteinander verknüpft werden. Damit bildet das Verknüpfen von Lösungsabbildungen (Join) eine wesentliche Operation bei der Evaluation einer SPARQL-Anfrage.

Zunächst sei angemerkt, dass sich eine Multimengen von Lösungsabbildungen gut in tabellarischer Form repräsentieren lässt, wobei eine Zeile in der Tabelle genau einer Lösungsabbildung entspricht. Da ein Join-Operator zwei Operatoren miteinander verknüpft, die Multimengen von Lösungsabbildungen produzieren, ergeben sich Parallelen mit der Evaluation von Join-Operationen in relationalen Datenbanksystemen. Aus diesem Grund können die in RDBMS gängigen Join-Algorithmen ebenfalls Ausführungsstrategien im Kontext der Beantwortung von SPARQL-Anfragen.

Obwohl eine Vielzahl von Join-Algorithmen in der Literatur beschrieben werden, kommen in kommerziellen Systemen wie IBM DB2, Oracle Database und Microsoft SQL Server im Wesentlichen Block-Nested-Loop-Join, Sort-Merge-Join und Hash-Join zum Einsatz[7] [Mul03, Mic13][CA+13, Abschn. 11.3]. Da das RCS ebenfalls die Daten auf Seiten aufgeteilt abspeichert, lassen sich die genannten Join-Algorithmen und auch die Kostenabschätzungen übertragen. Analog wie in einem relationalem DBMS muss im RCS der Optimierer anhand der zu erwartenden partizipierenden Tripel über den inneren und äußeren Operator[8] entscheiden. Die Ausführungskosten für diese Algorithmen werden in Abschnitt 5.6 näher betrachtet.

[7] In bestimmten Anwendungsfällen verwendet IBM DB2 einen Zigzag-Join [CS13] und Oracle Clustered-Join [LR+99].

[8] Vergleichbar zur inneren und äußeren Relation bei RDBMS

5.4.6 Weitere Ausführungsstrategien

Die vorherigen Abschnitte beschreiben vor allem Ausführungsstrategien, die spezifisch für den Resource Centered Store sind. Diese alleine reichen jedoch nicht aus, um die Lösungsabbildungen für eine Anfrage zu berechnen. Darüber hinaus werden noch weitere einfache Operatoren und Strategien verwendet, die in diesem Abschnitt nur überblicksartig beschrieben werden.

- *Sortierung* – Erstellen einer Sequenz von nach spezifizierten Kriterien sortierten Lösungsabbildungen (`ORDER BY`)
- *Duplikatentfernung* – Entfernen aller Duplikate in einer Multimenge von Lösungsabbildungen (`DISTINCT`)
- *Teilsequenz* – Extraktion einer Teilsequenz aus einer Sequenz von Lösungsabbildungen (`LIMIT`, `OFFSET`)
- *Pipelining* – Weiterleiten einer Lösungsabbildung an den nachfolgenden Operator sobald wie möglich
- *Scan* – Sequentielles Lesen aller Datenseiten ohne Verwendung von Subjekt-, Prädikat- oder Objektindex

Betrachtet man das Forschungsgebiet der Anfragebearbeitung in relationalen Datenbanksystemen, dann ließen sich noch weitere Strategien zur effizienteren Ausführung auf die Anfragebearbeitung im RCS übertragen und untersuchen (z. B. Wiederverwendung von Zwischenergebnissen). Eine Betrachtung dieser weitergehenden Strategien ist jedoch außerhalb des Fokus' dieser Arbeit, da der Schwerpunkt auf einer grundlegenden Evaluation des vorgeschlagenen Speichermodells liegt.

5.5 Generation eines Ausführungsplans

Wie bereits zu Beginn dieses Kapitels ausgeführt, sind die wesentlichen Aspekte bei der Anfrageoptimierung die Definition des Suchraumes, einer Kostenfunktion und eines Enumerierungsalgorithmus'. Der Suchraum wird durch die im vorherigen Abschnitt beschriebenen Transformationsregeln und Ausführungsstrategien, die bislang unabhängig voneinander betrachtet worden sind.

Im Folgenden werden zunächst Hypothesen und Heuristiken formuliert, die zu einer verbesserten Grundlage für die Generierung von Ausführungsplänen führen. Anschließend wird ein Algorithmus angegeben, um die viel versprechensten Ausführungspläne im Suchraum zu traversieren. Dabei werden Heuristiken verfolgt die zum einen die Charakteristika das Speichermodells des Resource Centered Store und zum anderen die Eigenschaften des RDF-Datenmodells berücksichtigt.

5.5.1 Vorbereitung des Anfragemodells

In diesem Abschnitt werden zunächst Heuristiken beschrieben, die eine gute Grundlage für das Generieren von alternativen Ausführungsplänen und die Wahl eines möglichst effizienten Ausführungsplans schaffen.

5.5.1.1 Zusammenführen von Graphmustern

Ausgehend vom Resource Centered Store sind die folgenden Eigenschaften des Speichermodells und des Gesamtsystems für die Generierung einer guten Basis für die Generierung von alternativen Ausführungsplänen von Bedeutung: (1) Gruppieren der RDF-Daten anhand ihres Subjekts und (2) die Definition von Indexen über den RDF-Daten.

Aufgrund der Eigenschaft (1) wird insbesondere die Auswertung von sternförmige Basis-Graphmuster durch das Speichermodell unterstützt. Diese Eigenschaft kann am besten während der Berechnung der Lösungen einer Anfrage ausgenutzt werden, wenn möglichst große Sternmuster durch einen Operator abgedeckt werden. Unter Umständen werden dabei optionale Graphmuster so platziert werden, dass sie gemeinsam mit einem Sternmuster ausgewertet werden können.

Wegen der Eigenschaft (2) können gegebenenfalls Indexe zur Berechnung von Teilergebnissen einer Anfrage herangezogen werden. Damit viele der existierenden Indexe für die Anfrage potentiell in Frage kommen – also zulässig sind, ist ein Zusammenführen von Basis-Graphmustern sinnvoll. Dadurch steigt die Wahrscheinlichkeit, dass Teile einer Anfrage durch einen Indexmuster überdeckt werden.

Die beiden Betrachtungen legen die Hypothese nahe, dass ein weitestgehendes Zusammenführen von Basis-Graphmustern zu einer günstigeren Ausgangssituation führt. Bei der Generierung von Ausführungsplänen werden daher zunächst die Transformationsregeln angewendet, die zu größeren Basis-Graphmustern im Anfragemodell führen:

- Eliminieren von Join-Operatoren (Regel 1)
- Distributivität von Vereinigung und Join (Regel 10)
- Verschieben von Filterausdrücken (Regel 6)

Beim Zusammenführen von Graphmustern mittels der Transformationsregel 1 werden insbesondere auch Join-Operationen mit leerem Graphmuster aus dem Anfragemodell eliminiert. Darüber hinaus werden redundante Tripelmuster in Join-Operationen entfernt.

5.5.1.2 Verschieben von Filterausdrücke

Nachdem durch die Anwendung der genannten Transformationsregeln die Basis-Graphmuster weitestgehend zusammengeführt worden sind, können die Filterausdrücke logisch optimiert werden. Um die Zusammenhänge von größeren Filterausdrücken untersuchen zu können, werden diese mittels der Transformationsregeln 3 und 4 zusammengeführt. Anschließend können beispielsweise bei gleichgesetzten Variablen eine von diesen entfernt werden. Dies reduziert zum einen die Anzahl der zu betrachtenden Variablen und ermöglicht unter Umständen Erkennen von größeren Sternmustern. Des Weiteren können auch Teile der Filterausdrücke oder der Anfrage aufgrund von Tautologien, Kontradiktionen oder Inklusion von Filterausdrücken eliminiert werden.

Des Weiteren werden Filterausdrücke dahingehend untersucht, ob sie eine mit einer Konstanten gleichgesetzte Variable enthalten. Falls die Variable

in keinem anderen Prädikat des Filterausdrucks verwendet wird, kann in der Anfrage die Variable durch die Konstante ersetzt werden. Dadurch konkretisieren sich im weiteren Verlauf der Optimierung die Kostenabschätzungen für die Auswertung der Graphmuster.

Wie bei der Optimierung in relationalen DBMS ist eine frühzeitige Auswertung von Filterausdrücken wünschenswert, damit im weiteren Verlauf weniger große Zwischenergebnisse verarbeitet werden müssen. Mittels der Transformationsregeln 5 und 6 können Filterausdrücke zunächst geeignet[9] zerlegt und weiter nach unten in dem Anfragemodell bewegt werden.

5.5.2 Enumerierungsalgorithmus

In dem nächsten Schritt werden alternative Ausführungspläne generiert und anhand einer Kostenfunktion bewertet. Die verschiedenen Ausführungspläne ergeben sich zum einen dadurch, dass zwischen den im Abschnitt 5.4 vorgestellten Ausführungsstrategien gewählt werden kann. Zum anderen können die Graphmuster einer Anfrage in unterschiedlicher Reihenfolge miteinander verknüpft werden. Das Verknüpfen von Teilergebnissen stellt dabei einen wesentlichen Faktor für die Optimierung der Ausführungszeit dar, da Join-Operationen den Hauptanteil der Kosten bei der Anfrageausführung verursachen. Daher wird im Folgenden vor allem auf die Optimierung der Join-Reihenfolge eingegangen.

5.5.2.1 Dynamische Programmierung

Da die Anzahl der möglichen Kombinationen exponentiell mit der Anzahl der Basis-Graphmuster wächst, ist ein betrachten aller denkbaren Kombinationen während der Anfrageoptimierung nicht möglich. Im schlimmsten Fall müssten alle Join-Kombinationen von einzelnen Tripelmustern betrachtet werden. Angesichts der über 17 Milliarden Möglichkeiten bei nur zehn Tripelmustern wurde der Ansatz der dynamischen Programmierung als Enumerierungsalgorithmus für mögliche Ausführungspläne gewählt, wie er in relationalen Datenbanken Anwendung findet [SAC+79].

In [NW10] wird darüber hinaus festgestellt, dass Anfragen häufig mehrere sternförmige Teile beinhalten und daher nicht alle möglichen Kombinationen des Verknüpfens von sternförmigen Graphmustern überprüft werden können. Da jedoch im RCS sternförmige Graphmuster als einzelne Operatoren im Anfragemodell dargestellt werden und somit die Reihenfolge der Tripelmuster in einem Sternmustern unerheblich ist, verkleinert sich in Abhängigkeit von den Größen der Sternmuster außerdem der Suchraum der Ausführungspläne.

Im Kontext der Optimierung von SPARQL besteht der erste Schritt bei der Optimierung per dynamischer Programmierung in dem Wählen des besten Ausführungsstrategie für die Basis-Graphmuster. Dabei wird die beste Ausführungsstrategie anhand einer Kostenfunktion bestimmt. Im nächsten Schritt werden Zweier-Kombinationen der zuvor bestimmten besten Ausführungsstrategien betrachtet und wiederum die beste Join-Reihenfolge bestimmt. Der darauf folgende Schritt betrachtet dann Dreier-Kombination von Graphmustern. In den weiteren Schritten wird jeweils ein weiteres Graphmuster solange hinzugenommen, bis alle Graphmuster betrachtet worden sind.

[9] Unter Berücksichtigung der Verwendung der Variablen in den Operatoren

Selbst das Prinzip der dynamischen Programmierung kann bei vielen Join-Partnern zu zeitaufwändig sein (Laufzeitkomplexität von $O(3^n)$). Um die Anzahl der zu betrachtenden Ausführungspläne frühzeitig weiter zu reduzieren, kann das Verfahren *greedy join enumeration* [Loh88] verwendet werden Laufzeitkomplexität von $O(n^3)$. Bei diesem Verfahren wird in jeder Iteration der beste Ausführungsplan ausgewählt und nur dieser im nachfolgenden Schritt betrachtet.

Der initiale Schritt bei der Enumeration der Ausführungspläne im Falle der Optimierung von SPARQL-Anfragen im RCS unterscheidet sich von dem bei relationalen Datenbanken. Während bei RDBMS die Anzahl der zu verknüpfenden Relationen feststeht, kann sich die Anzahl der zu betrachten Graphmuster in Abhängigkeit vom gewählten Zugriffspfad ändern. Die Ursache liegt dabei in den verschiedenen Ausführungsstrategien für die Evaluation eines Basis-Graphmusters:

- Zerlegung in sternförmige Graphmuster (Strategien 1 und 2)
- Zerlegung aufgrund von Indexen (Strategie 7)
- Zerlegung in sternförmiges Graphmuster mit gleichzeitiger Auswertung eines optionalen Graphmusters (Strategie 3)

Betrachtet man die ersten beiden Möglichkeiten, dann kann das Basis-Graphmuster je nach angewendeter Ausführungsstrategie in unterschiedliche Teilmuster zerfallen. Im Allgemeinen muss nämlich das in dem Basis-Graphmuster enthaltene Sternmuster nicht mit dem Graphmuster eines zulässigen Index übereinstimmen. Dementsprechend entstehen beim Zerlegen des Basis-Graphmusters sowohl ein Join-Operator als auch ein weiteres Basis-Graphmuster.

Abbildung 5.8 zeigt zwei mögliche Zerlegung eines Basis-Graphmusters. Beim Ausführungsplan A) wird das Graphmuster in zwei sternförmige Teilmuster zerlegt, deren Lösungen anschließend über einen Join miteinander verknüpft werden. Im Gegensatz dazu wird beim Plan B) ein Index über den ersten drei Tripelmustern verwendet. Darüber hinaus können je Plan können zum einen noch weitere Ausführungsstrategien (vgl. Abschnitt 5.4) betrachtet werden. Zum anderen verändert sich die Eingabe für den nächsten Schritt der dynamischen Programmierung.

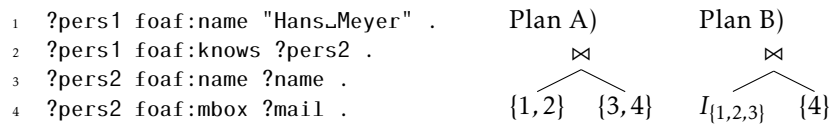
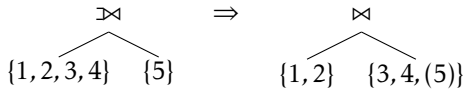


Abbildung 5.8: Zwei Varianten für die Ausführung eines Basis-Graphmusters: Plan A) Sternmuster und Plan B) Index

Bei der nächsten anwendbaren Ausführungsstrategie wird nicht nur der Basis-Graphmusteroperator betrachtet, sondern auch noch ein damit verknüpftes optionales Graphmuster. D.h. im Vergleich zur Optimierung in relationalen

Datenbanksystemen müssen im initialen Schritt der dynamischen Optimierung gleichzeitig mehrere Operatoren untersucht werden, um die anfänglich besten Ausführungsstrategien zu ermitteln.

Wenn beispielsweise ein Basis-Graphmuster wie in Abbildung 5.8 gegeben ist und zusätzlich ein Optionales Graphmuster 5: `OPTIONAL {?pers2 foaf:plan ?plan }` existiert, dann könnten die Kosten für die gleichzeitige Auswertung von $\{3,4\}$ und 5 günstiger sein (Strategie 3), als eine Verwendung eines Indexes über den Tripelmustern $1, 2$ und 3.



Neben der Tatsache, dass eine Graphmuster durch die Verwendung eines Indexes in Teilgraphen zerfällt, können aus der Menge der über den RDF-Graphen definierten Indexen kein, ein oder mehrere Indexe zur Beantwortung von Teilen einer Anfrage herangezogen werden. Dazu wird zunächst die Menge der zulässigen Indexe (vgl. Abschnitt 4.3.1) bestimmt, aus denen sich im Anschluss die überdeckenden Indexmengen ergeben. Aufgrund dieser entstehen verschiedene initiale Ausführungspläne, deren Kosten zu berechnen sind.

5.5.2.2 Heuristiken zum Generieren von alternativen Ausführungsplänen

Zum Generieren von alternativen Ausführungsplänen werden Heuristiken auf den anfänglichen Ausführungsplan angewendet. Für diese werden anschließend die Kosten für die Ausführung berechnet. Die Menge der alternativen Ausführungspläne besteht zum einen aus einem Ausführungsplan ohne Indexzugriff und zum anderen aus den Ausführungsplänen, die auf mindestens einen Index zugreifen. Auf die generierten Ausführungspläne werden im Anschluss Heuristiken angewendet, damit die selektiveren Operatoren zuerst ausgeführt und mit anderen Operatoren verknüpft werden.

Generierung von alternativen Operatorbäumen Bei der Generierung des Ausführungsplans ohne Indexzugriff bilden sternförmige Basis-Graphmuster den Ausgangspunkt. Jedes Graphmuster des Anfragemodells wird daraufhin überprüft, ob es ein Sternmuster enthält. Dabei werden insbesondere auch optionale Graphmuster miteinbezogen, da diese gegebenenfalls gleichzeitig ausgewertet werden können (vgl. Ausführungsstrategie 3). Wenn ein Graphmuster einen sternförmigen Teil beinhaltet, wird dieses dementsprechend zerlegt und durch einen Join-Operator ersetzt (Transformationsregel 2).

Um die Ausführungspläne mit Indexzugriff generieren zu können, werden zunächst die zulässigen Indexe sowie die möglichen Überdeckungen mit den Graphmustern in der Anfrage bestimmt (vgl. Abschnitt 4.3.1). Entsprechend der Überdeckungen wird anschließend die Anfrage durch Anwenden der Regel 2 zerlegt. Das heißt, ein Graphmuster wird in ein überdecktes und ein nicht überdecktes Teilgraphmuster aufgeteilt und mit einer Join-Operation verknüpft. Im letzten Schritt wird wie zuvor beschrieben eine Zerlegung bezüglich der enthaltenen Sternmuster vorgenommen.

Die Anzahl der entstehenden Ausführungspläne hängt direkt von der Anzahl der zulässigen Indexe und den möglichen Überdeckungen ab. Wenn für eine Anfrage N zulässige Indexe existieren und jeder genau einmal verwendet werden könnte, dann würden mit diesem Verfahren 2^N Ausführungspläne

generiert werden. Da SPARQL-Anfragen im Allgemeinen aus kleineren Basis-Graphmustern bestehen, werden nur wenige Indexe zulässig sein und jeder nur einmal pro Anfrage verwendet. Darüber hinaus kann nicht jeder Index mit allen anderen kombiniert werden, wodurch die Anzahl an entstehenden Ausführungspläne nochmals reduziert[10] wird.

Wie im Abschnitt „Indexierte Graphmuster" auf Seite 106 beschrieben wurde, können im RCS gleichzeitig mit dem Indexzugriff auch Filterausdrücke ausgewertet werden. Durch das Hinzunehmen von Filterausdrücken zu dem Indexzugriff können weitere Varianten eines Ausführungsplanes generiert werden. Dazu muss gegebenenfalls ein Filterausdruck mittels der Transformationsregel 5 in mehrere Teile aufgespalten werden.

Ausführungsreihenfolge Auf jeden generierten Ausführungsplan wird eine Heuristiken angewandt, um eine gute initiale Ausführungsreihenfolge zu bestimmen. Die Heuristik basiert auf der Anzahl der konstanten Komponenten in den Graphmustern. Dabei wird eine Komponente ebenfalls als konstant angesehen, wenn es einen zugehörigen Filterausdruck gibt, in dem eine Variable mit konstanten Werten gleichgesetzt wird. Dies ermöglicht dem Optimierer eine der Ausführungsstrategien 4 oder 5 (s. Seiten 102ff.) zu verwenden. Nachfolgend werden diese Variablen als *Quasi-Konstanten* bezeichnet.

Die Anfrage in Abbildung 5.9 enthält beispielsweise die quasi-konstante Variable `?name`. Da die Variable nur zwei Werte annehmen kann, ist es effizienter die Anfrage mittels der Strategie 5 auszuwerten, anstatt die Datenseiten nur anhand der Prädikate `foaf:name` und `foaf:knows` zu filtern.

```
SELECT ?other WHERE {
    ?pers foaf:name ?name .
    ?pers foaf:knows ?other.
    FILTER (?name == "Meyer" || ?name = "Schmidt") .
}
```

Abbildung 5.9: Wertung der Variable `?name` als Konstante

Anhand der nachfolgenden Priorisierung wird die Reihenfolge der Join-Operatoren in einem Ausführungsplan festgelegt.

i) Sternmuster mit (quasi-)konstantem Subjekt, wobei größere höher gewichtet werden

ii) Einzelnes Tripelmuster mit (quasi-)konstanten Subjekt

iii) Indexzugriff (ggf. mit gleichzeitiger Auswertung von Filterausdrücken)

iv) Sternmuster mit einer hohen Anzahl an (quasi-)konstanten Prädikaten und Objekten

v) übrige Graphmuster

[10] Ein Konflikt schließt bereits ein Viertel der Kombinationsmöglichkeiten aus; ein weiterer würde ein weiteres Viertel eliminieren.

Im Wesentlichen wird die Prioritätenliste durch das Speichermodell des RCS bestimmt. Da zum Auswerten eines Graphmuster und Tripelmuster mit gegebenem Subjekt nur eine Datenseite gelesen werden muss, können diese Teile einer Anfrage sehr effizient beantwortet werden. Ein Sternmuster wird dabei als höherwertig angesehen, da ein größerer Teil der Anfrage abgedeckt wird.

Die nächst niedrigere Priorität haben Indexzugriffe, weil davon ausgegangen wird, dass ein sequentielles Lesen der Indexeinträge im Allgemeinen effizienter als das auswerten eines Sternmusters ohne gegebenen Subjekt ist.

Anschließend werden die übrigen Sternmuster betrachtet, wobei im Wesentlichen die Anzahl der darin enthaltenen Konstanten ausschlaggebend ist. Die Hypothese dahinter ist, dass mit steigender Anzahl an Konstanten über die Prädikat- und Objektindexe die Anzahl der zuzugreifenden Datenseiten stärker eingeschränkt werden kann. Darüber hinaus werden Tripelmuster mit konstantem Objekt höher gewichtet als die mit gegebenem Prädikat. Dies liegt in der Hypothese begründet, dass konstante Objekte im Allgemeinen eine höhere Selektivität besitzen. Wie Tsialiamanis in [TSF+12] argumentiert, wird auch in dieser Arbeit angenommen, dass Tripelmuster mit einem Literal als Objekt selektiver sind und somit eher ausgewertet werden sollten. Eine Höhergewichtung von Tripeln mit gleichzeitig gegebenen Prädikat und Objekt ist im Vergleich zu anderen nativen RDF-Datenbanksystemen nicht sinnvoll, da die Prädikat- und Objektindexe nicht festgestellt werden kann, ob ein Tripel der Form $(?po)$ in der Datenbank existiert. Quasi-Konstanten haben im Vergleich zu Konstanten ein geringeres Gewicht, wobei auch die Anzahl der möglichen Werte für eine Variable berücksichtigt wird.

Zuletzt werden die restlichen Graphmuster in einen Ausführungsplan integriert, da für deren Auswertung mehrere Join-Operationen ausgeführt werden müssen. Die Reihenfolge des Verknüpfens der Tripelmustern wird heuristisch wie folgt priorisiert: $(?po), (??o), (?p?), (???)$. Im Gegensatz zu zuvor werden Tripelmuster mit gleichzeitig gegebenen Prädikat und Objekt höher gewichtet, da diese Kombination wahrscheinlich eine höhere Selektivität besitzt, als wenn nur Prädikat oder Objekt konstant ist. Wie zuvor haben auch in diesem Fall Quasi-Konstanten eine geringeres Gewicht.

Verknüpfen von Operatoren Bei der Initiierung wurden basierend auf Heuristiken eine Menge von Ausführungsplänen erstellt, die die Grundlage für das Bestimmen des auszuführenden optimalen Ausführungsplan bilden. Für jeweils ein Paar von Operatoren werden die möglichen Ausführungsstrategien untersucht, die beste Strategie anhand der Kostenfunktion bestimmt und an den nächsten Schritt der dynamischen Programmierung weitergegeben. Dieser Vorgang wiederholt sich solange, bis die Join-Reihenfolge für alle Operatoren festgelegt worden ist.

Dazu wird jedem Operator in einem Ausführungsplan zunächst eine Ausführungsstrategie initial zugewiesen. Für jede zuvor besprochene Gruppe von Operatoren wird nachfolgend die initiale Strategie aufgelistet; gegebenenfalls gibt der Wert in Klammern die Strategie für den Fall von Quasi-Konstanten an (vgl. Abschnitt 5.4).

- Sternmuster bzw. Tripelmuster mit konstantem Subjekt: Strategie 1 (Strategie 4)

- Indexzugriff: Strategie 7
- beliebiges Sternmuster oder Tripelmuster: Strategie 2 (Strategie 5)
- Join-Operation: Nested-Loop-Join; Sort-Merge-Join, falls beide eingehenden Iteratoren bzgl. der Join-Variable sortierte Lösungsabbildungen liefern

Im Wesentlichen wird in der Phase der dynamischen Programmierung darüber entschieden, welcher Operator die innere „Relation" bei einem Join bildet. Die Wahl der inneren Relation ist dabei von dem verwendeten Join-Algorithmus sowie der Größe der Lösungsmengen bzw. der Zwischenergebnisse abhängig. Dieser Punkt wird im Abschnitt 5.6 eingehender diskutiert. Die Veränderung der Join-Reihenfolge kann aufgrund der Kommutativität der Join-Operation und durch Anwendung der Transformationsregel 9 erreicht werden.

Bei der Untersuchung der Join-Reihenfolge können drei Formen von Join-Bäumen unterschieden werden: linksseitig tiefe, rechtsseitig tiefe und buschige Bäume. Neben der im Vergleich zur Betrachtung von buschigen Bäumen geringeren Anzahl an Varianten sind linksseitig tiefe Bäume vorteilhaft, weil (i) sie einen geringeren Suchraum aufspannen, (ii) die kostenreduzierende Technik des Pipelinings eingesetzt werden kann, (iii) der Hauptspeicherbedarf tendenziell geringer ausfällt und (iv) bei Verwendung des Nested-Loop-Joinalgorithmus' keine Zwischenergebnisse auf dem Sekundärspeicher abgelegt werden müssen. [SMK97]

Beispiel 5.3. Wenn beispielsweise für die drei Operatoren $\mathsf{op}_1, \mathsf{op}_2, \mathsf{op}_3$ und op_4 die besten Ausführungspläne bestimmt worden sind, werden für $\mathsf{op}_1 \bowtie \mathsf{op}_2 \bowtie \mathsf{op}_3 \bowtie \mathsf{op}_4$ die folgenden Varianten betrachtet:

- $\mathsf{op}_i \bowtie \mathsf{op}_k$ und $\mathsf{op}_k \bowtie \mathsf{op}_i$ für $i,k = 1 \ldots 4, i \neq k$
- $opt(\mathsf{op}_i \bowtie \mathsf{op}_k) \bowtie \mathsf{op}_l$ für $i,k,l = 1 \ldots 4, i \neq k \neq l$
- $opt(\mathsf{op}_i \bowtie \mathsf{op}_k \bowtie \mathsf{op}_k) \bowtie \mathsf{op}_m$ für $i,k,l,m = 1 \ldots 4, i \neq k \neq l \neq m$

□

5.5.3 Nachbearbeitung des Ausführungsplans

Nachdem die günstigste Join-Reihenfolge bestimmt worden ist, findet eine Nachbearbeitung des Ausführungsplans statt. Bei der Ausführung eines Operators können Variablenbindungen bereitgestellt werden, die von späteren Operatoren nicht benötigt werden. Durch Anwenden der Transformationsregel 7 auf jeden Operator werden im weiteren Verlauf unbenötigte Variablenbindungen möglichst früh aus den Iteratoren entfernt. Damit wird sich die zu transferierende Datenmenge kleiner und die Größen von Zwischenergebnissen reduziert. Diese Modifikation des Ausführungsplans kann nicht vor dem Bestimmen der Join-Reihenfolge ausgeführt werden, da durch Verwenden von Indexen oder bestimmten Ausführungsstrategien (z. B. Strategie 1) Graphmuster-Operatoren zerlegt werden. Damit ist vorher nicht feststellbar, welche Variablenbindungen von den Iteratoren bereitgestellt werden müssen.

5.6 Kostenmodell

Im Resource Centered Store setzen sich die Kosten eines Ausführungsplans aus den IO-Kosten und den Join-Kosten zusammen. Erstere werden durch die Kosten für das Laden der relevanten Datenseiten vom Sekundärspeicher dominiert. Letztere werden durch die Größe der Zwischenergebnisse bestimmt. Im Folgenden werden zunächst Strategien zur Abschätzung der IO-Kosten und dann Abschätzungen für die Zwischenergebnisgrößen betrachtet.

5.6.1 IO-Kosten

IO-Kosten sind nur für diejenigen Operationen eines Anfrageausführungsplans relevant, die einen Zugriff auf den Sekundärspeicher erfordern. Dies sind zum einen Graphmuster-Operatoren und zum anderen Join-Operatoren, die mindestens einen Graphmuster-Operator als Eingabe haben. Die IO-Kosten für einen Operator op werden im Folgenden als $|\mathsf{p}_{\mathsf{op}}|$ notiert. Für alle anderen Operatoren können die IO-Kosten mit 0 angesetzt werden. Die Kosten für den Zugriff auf Indexe wurden bereits in den Abschnitten 4.5 diskutiert.

Wie im Abschnitt „Operationen" auf Seite 33 dargestellt wurde, sind die Kosten für den Zugriff auf den Sekundärspeicher von den im Tripelmuster enthaltenen Konstanten abhängig. Während ein gegebenes Subjekt aufgrund des gewählten Speichermodells konstante Zugriffskosten verursacht, bestimmt bei gegebenem Prädikat oder Objekt die Verteilung der relevanten Tripel auf den Datenseiten die Zugriffskosten. Zunächst werden Strategien zur Abschätzung der zugegriffenen Datenseiten für ein Tripelmuster untersucht. Daraus lassen sich Aussagen über beliebige Graphmuster ableiten.

Tripelmuster. Da bei gegebenen Subjekt – unabhängig von den anderen Komponenten des Tripelmusters – genau eine Datenseite zugegriffen werden muss, müssen nur die Fälle $(?, p, ?)$, $(?, ?, o)$ und $(?, p, o)$ untersucht werden. In den ersten beiden Fällen wird die Anzahl der zugegriffenen Datenseiten auf Basis von zuvor erstellten Histogrammen bestimmt. Die Histogramme speichern die Anzahl der Datenseiten, die Tripel mit einem bestimmten Prädikat oder Objekt enthalten. Um die Größe der Histogramme zu reduzieren werden diese zu Equi-Width-Diagramme transformiert.

Für ein gegebenes Prädikat kann diese Information aus dem Prädikatindex abgeleitet werden, da dieser zu jedem Prädikat die physischen Adressen derjenigen Subjekte speichert, die zusammen mit diesem in mindestens einem Tripel auftreten (vgl. Abschnitt „Zugriffsstrukturen" auf Seite 32). Um die zuzugreifenden Datenseiten für ein Prädikat zu erhalten, wird für jede Position eines gesetzten Bits der Slot mittels einer entsprechenden Bitmaske ausgeblendet und anschließend die doppelten Seiten-IDs entfernt. Die Menge der für ein Prädikat t^p zuzugreifenden Datenseiten wird als $|\mathsf{p}_{t^p}|$ notiert. Analog kann das Histogramm für die Objekte aus dem Objektindex erstellt werden und die Menge der Datenseiten wird mit $|\mathsf{p}_{t^o}|$ bezeichnet.

Ein detailliertes Histogramm für ein Tripelmuster der Form $(?, p, o)$ kann erstellt werden, indem die Bitsets des jeweiligen Prädikats und Objekts verundet wird und anschließend mit zuvor beschriebenen Verfahren die zuzugreifenden Datenseiten bestimmt werden. Jedoch ist dieses aufgrund der Anzahl an Kombinationsmöglichkeiten von Prädikat- und Objektwerten ($O(|t^p| \cdot |t^o|)$)

recht aufwendig und nicht praktikabel. Daher wird die Anzahl der zuzugreifenden Datenseiten auf Basis der Histogramme für Prädikate und Objekte berechnet. Sei $\mathcal{D}$ eine Datenbank und $|\mathcal{D}|$ die Anzahl der belegten Datenseiten, dann wird die Anzahl wie folgt berechnet:

$$|\mathsf{p}_{t^p t^o}| = \frac{|\mathsf{p}_{t^p}| \cdot |\mathsf{p}_{t^o}|}{|\mathcal{D}|}$$

Zusammengefasst ergibt sich die folgende Formel, um für ein beliebiges Tripelmuster die Anzahl der zuzugreifenden Daten zu bestimmen:

$$|\mathsf{p}_t| = \begin{cases} t^s \in \mathbb{I} & 1 \\ sonst & \frac{|\mathsf{p}_{t^p}| \cdot |\mathsf{p}_{t^o}|}{|\mathcal{D}|} \end{cases}$$

Sternförmige Graphmuster. Die Berechnung der Lösungen für ein sternförmiges Graphmuster ähnelt der eines Tripelmusters, da aufgrund des Speichermodells ebenfalls keine Join-Operation erforderlich ist. Da dadurch die für das Berechnen der Lösung relevanten Tripel auf einer Datenseite gruppiert sind, wird die Anzahl der zu lesenden Datenseiten über das Tripel mit den wenigsten zuzugreifenden Datenseiten bestimmt. Für ein sternförmiges Basis-Graphmuster $P \in \mathsf{BGP}$ mit $P = \{t_1, \ldots t_n\}, t_i \in \mathcal{T}$ ergibt sich somit die folgende Abschätzung:

$$|\mathsf{p}_P| = \min_{i=1,\ldots n} |\mathsf{p}_{t_i}|$$

Beliebige Graphmuster. Zum Berechnen der Lösungsabbildungen eines beliebigen Graphmuster wird dieses in einen Join-Baum bestehend aus Stern- und Tripelmustern zerlegt. Stern- und Tripelmuster bilden die atomaren Operanden, da für diese keine Join-Operation ausgeführt werden müssen. Beim Betrachten einer Join-Operation können zwei Situationen unterschieden werden:

(i) Die Lösungen beider Operanden müssen vom Sekundärspeicher gelesen werden.

(ii) Nur die Lösungen eines Operanden müssen vom Sekundärspeicher gelesen werden, die des anderen liegen im Hauptspeicher vor.

Zu (i): Diese Situation tritt auf, wenn die Lösungen für die Operanden noch nicht berechnet worden sind oder wenn es sich um auf den Sekundärspeicher ausgelagerte Zwischenergebnisse handelt. Da das RCS-System die Daten wie auch relationale DBMS auf Datenseiten verwaltet, können die Abschätzungen für Join-Operationen aus der relationalen Welt übernommen werden. Für die drei in dieser Arbeit verwendeten Join-Algorithmen Nested-Loop-Join, Sort-Merge-Join und Hash-Join ergeben sich die Abschätzungen wie sie in Tabelle 5.1 aufgelistet sind. Dabei bezeichnet L und R die beiden zu verknüpfenden Graphmuster und m die Größe des Hauptspeichers.

Bei einem Sort-Merge-Join können sich die IO-Kosten um die Sortierkosten $2|\mathsf{p}_{t_l}| \cdot \left\lceil \log_m |\mathsf{p}_{t_l}| \right\rceil$ reduzieren, falls das Zwischenergebnis für t_l sortiert vorliegt.

Join-Algorithmus	**Seitenzugriffe** = $\lvert p_{\bowtie} \rvert$
BNL-Join	$\lvert p_L \rvert + \frac{\lvert p_L \rvert \cdot \lvert p_R \rvert}{m-1}$
Sort-Merge-Join	$\lvert p_L \rvert + \lvert p_R \rvert + 2\,(\lvert p_L \rvert \cdot \lceil \log_m \lvert p_L \rvert \rceil + \lvert p_R \rvert \cdot \lceil \log_m \lvert p_R \rvert \rceil)$
Hash-Join	$3(\lvert p_L \rvert + \lvert p_R \rvert)$

Tabelle 5.1: Seitenzugriffe für verschiedene Join-Algorithmen [ITL10]

Zu (ii): In der Situation wurde o.B.d.A. das Ergebnis des Operators R bereits berechnet und kann vollständig im Hauptspeicher gehalten werden. Unter Verwendung eines Nested-Loop-Joins werden die IO-Kosten von der Anzahl der zuzugreifenden Datenseiten für L bestimmt, da jede zugehörige Datenseite genau einmal geladen werden muss. Die Kosten können somit mit $\lvert p_{\bowtie} \rvert = \lvert p_L \rvert$ abgeschätzt werden.

Anstatt alle Lösungen für das Graphmuster L zu berechnen können in dieser Situation auch die Ausführungsstrategien 9 und 10 (vgl. Seiten 110 ff.) angewendet werden. Bei diesen Strategien werden die Lösungen von R genutzt, um über Subjekt-, Prädikat- und Objektindex die Menge der zuzugreifenden Datenseiten weiter zu reduzieren. Die tatsächlichen Kosten dieser Ausführungsstrategien kann im Vorhinein nur unzureichend bestimmt werden, da sie direkt vom Zwischenergebnis von R abhängen. Der Einsatz dieser Strategien ist für eine Lösung μ von R sinnvoll, wenn für das aus der Substitution der Variablen in L resultierende Graphmuster weniger Datenseiten zugegriffen werden müssen, d.h. wenn $\lvert p_{\mu[L]} \rvert \ll \lvert p_L \rvert$ für ein $\mu \in \Omega_R$ gilt.

5.6.2 Selektivität und Größe von Zwischenergebnissen

Die Größe eines Zwischenergebnisses von einem Operator ist über die Anzahl der von diesem generierten Lösungsabbildungen definiert. Um die Größen der Zwischenergebnisse abschätzen zu können, werden Informationen über die Selektivitäten von Tripelmustern, sternförmigen Graphmustern und Filterausdrücken benötigt. In der Literatur wurden im Wesentlichen zwei Varianten für das Abschätzen der Selektivität von Tripelmuster und der Ergebnisgröße von Join-Operationen vorgeschlagen: Statistische Informationen und Data-Mining.

Statistische Verfahren. In diesem Fall werden für die RDF-Daten Histogramme erzeugt, in denen die Anzahl der Vorkommen von Ressourcen und Literalen als Subjekt, Prädikat und Objekt festgehalten wird. In [SSB+08] beschreiben Stocker et al., wie sie mittels Histogrammen über Subjekt, Prädikat und Objekt die Kardinalitäten der Operatoren abgeschätzt werden. Da sie unrealistischerweise die Unabhängigkeit von Subjekt, Prädikat und Objekt annehmen, sind die auf einzelnen Komponenten eines Tripels basierenden Abschätzung ungenau.

Data Mining. Den durch Unabhängigkeitsannahme entstehenden Fehler bei der Kardinalitätsabschätzung wird in der Literatur durch Verfahren des Data Minings begegnet. In [V+10] wurde das Erheben von Stichproben vorgeschlagen, um die Kosten und die Kardinalitäten für eine Anfrage abschätzen zu

können. Dieses Herangehen setzt jedoch voraus, dass eine repräsentative Teilmenge der RDF-Daten bestimmt werden kann. Vidal et al. beschreiben das Vorgehen auf einem hohen Abstraktionsniveau; auf das Vorgehen zum Bestimmen der repräsentativen Daten wird nicht näher eingegangen, sondern als gegeben vorausgesetzt.

Die Graph-Mining-Techniken Maximum-Dependency-Tree und Pattern-Tree werden in [MASS08] zum Abschätzen der Kardinalitäten verwendet. Selbst für kleine Datenmenge entstehen jedoch hohen Laufzeiten und somit ist das Verfahren nicht praktikabel.

In [HL11] werden für sternförmige Teilgraphen Bayessche Netzwerke gelernt, um das gemeinsame Auftreten von Ressourcen und Literalen voraussagen zu können. Des Weiteren werden auch Histogramme über häufige Join-Pfade erhoben. Auf Grundlage dieser Daten erreichen die Autoren für Ihren Anfragensatz eine höhere Präzision bei den Kardinalitätsabschätzungen als den nachfolgend beschriebenen Ansatz. Da in der Arbeit nur sehr wenige (und von denen in [NW10] verschiedene) Anfragen zur Evaluation herangezogen wurden, sind die Aussagen dieser Arbeit nicht mit denen von Neumann et al. vergleichbar.

Hybride Verfahren. Neumann et al. erweitern die Histogramme über das Vorkommen von Ressourcen und Literalen um Informationen über Join-Partner [NW10]. Beispielsweise wird gespeichert, wie viele Join-Partner es für die Kombination Subjekt gleich Prädikat existieren. Darüber hinaus werden zusätzliche Informationen über häufige Join-Pfade (*engl. frequent paths*) gesammelt. Auf Grundlage dieser Datenbasis erzielen Neumann et al. eine höhere Präzision bei der Abschätzung der Kardinalitäten.

Das Vorgehen für Abschätzen der Kardinalitäten im Resource Centered Store basiert auf dem im RDF-3X eingesetzten Verfahren [NW10]. Da Histogramme über einzelnen Komponenten eines Tripels zu ungenauen Abschätzungen führt, haben Neumann et al. Histogramme über Zweier-Kombinationen der Tripelkomponenten erstellt. Im Gegensatz zu den dortigen sechs Histogrammen werden jedoch im RCS nur die folgenden drei Histogramme benötigt: SP für das Abschätzen der Muster ($s??$) und ($sp?$), OS für ($??o$) und ($s?o$) sowie PO für ($?p?$) und ($?po$). Die Tripelmuster (spo) und ($???$) müssen nicht gesondert betrachtet werden, da die Kardinalitäten entweder 1 oder der Gesamtzahl an Tripeln in der Datenbank entspricht.

Während im RDF-3X die Aggreationsindexe für das Erstellen der Histogramme verwendet werden können, können die Zugriffsstrukturen des RCS (Subjekt-, Prädikat- und Objektindexe) nicht verwendet werden. Zwar kann über die Prädikat- und Objektindexe die Anzahl der Subjekte ermittelt werden, mit denen ein bestimmtes Prädikat bzw. Objekt zusammen vorkommt, jedoch lässt sich die genaue Anzahl der Tripel nicht bestimmen. Dadurch ist ein Lesen aller Tripel – und damit aller Datenseiten – erforderlich (vgl. Abschnitt 3.2.3).

Zur Vereinfachung der Beschreibung werden im Folgenden die ins Histogramm einbezogenen Komponenten als erste und zweite Konstante bezeichnet. Zum Beispiel wäre im SP-Histogramm die erste Konstante S und die zweite Konstante P.

Beginn (s_b, p_b, o_b)		Ende (s_e, p_e, o_e)	
#Tripel			
#distinkte 1er-Komponente			
#distinkte 2er-Komponente			
Join-Partner bzgl. Subjekt			
	s=s	s=p	s=o
Join-Partner bzgl. Prädikat			
	p=s	p=p	p=o
Join-Partner bzgl. Objekt			
	o=s	o=p	o=o

Abbildung 5.10: Struktur eines Histogramm-Buckets nach [NW10]

Für das Erstellen eines Histogramms werden die zu berücksichtigenden Tripel zunächst nach dem Schlüssel (erste Konstante, zweite Konstante) sortiert. Pro Bucket in einem Histogramm werden die in Abbildung 5.10 dargestellten statistischen Informationen gespeichert. Die Werte in den ersten drei Zeilen werden für die Abschätzung der Kardinalitäten für ein einzelnes Tripelmustern benötigt. Der erste Wert gibt die Anzahl der Tripel in dem Bucket, der zweite und dritte Wert gibt die Anzahl der Tripel mit einer bzw. zwei Konstanten wieder. Die restlichen Daten dienen dem Abschätzen der Ergebnisgröße einer Join-Operation zwischen den Tripeln eines Buckets und aller Tripeln des RDF-Graphen. Falls der Join nicht genau über eine Komponente eines Tripels stattfindet, werden die Selektivitäten der der Join-Prädikate miteinander multipliziert. Das heißt, es wird eine Unabhängigkeit der Join-Prädikate angenommen.

Um die Defizite dieser Abschätzungen auszugleichen, werden in [NW10] zusätzlich Informationen über die häufigsten Join-Pfade (*engl. frequent path*) gespeichert. Von besonderem Interesse sind dabei sternförmige Tripelmengen und Pfade von Tripeln, bei denen das Objekt des einen Tripels das Subjekt des anderen ist. Im Gegensatz zum System RDF-3X werden im RCS die häufigsten Pfade in Form von sternförmigen Tripelmengen ohne Join- und Aggregationsoperationen während der Berechnung der Histogramme ermittelt. Von diesen ausgehend werden anschließend längere häufige Pfade bestimmt.

Darüber hinaus können im RCS die Statistiken der für eine Anfrage zulässigen Indexe für die Abschätzung der Selektivität von Teilmustern verwendet werden. Beispielsweise entspricht die Anzahl der Indexeinträge der Kardinalität für das Verknüpfen der überdeckten Tripelmuster. Für das Abschätzen der Kardinalität eines Teilmusters sind diese statistischen Informationen somit präziser als die zuvor beschriebenen. Selbst wenn das Indexmuster allgemeiner als das überdeckte Teilmuster der Anfrage ist, können die Statistiken des Indexes verwendet werden. Um die Join-Kardinalität des Teilmusters zu bestimmen, werden die Selektivitäten der spezielleren Tripelmuster der Anfrage mit der Anzahl der Indexeinträge multipliziert. Darüber hinaus können mittels der Histogramme bereits frühzeitig ungünstige Indexzugriffe mit einbezogenem Filterausdruck (Nachfiltern) aus den Ausführungsplänen eliminiert werden.

Beispiel 5.4. Der Index I in Abbildung 5.11 ist für die Anfrage zulässig, jedoch ist das Indexmuster allgemeiner als das überdeckte Graphmuster der Anfrage. Ein Indexzugriff würde sich lohnen, falls $p_I < p_Q$. Da alle Datenseiten des Indexes gelesen und die Lösungsabbildungen anschließend nachgefiltert werden, ist für die Entscheidung $p_I < p_Q$ die Selektivität des Filterprädikats `?project = ex:dbpedia` ausschlaggebend. Falls fast alle Indexeinträge zur Lösung von

Q gehören (geringe Selektivität) ist ein Indexzugriff effizienter, da diese kompakter gespeichert werden (vgl. Abschnitt 4.4). □

Index I (Kardinalität: 100k Lösungsabbildungen)

```
{ ?person foaf:currentProject ?project . }
```

Anfrage Q

```
SELECT * WHERE {
      ?person foaf:currentProject ex:dbpedia .
}
```

Abbildung 5.11: Selektivität von `ex:dbpedia` bestimmt Indexzugriff

Die Verwendung von statistischen Informationen über die häufigsten Pfade haben den Nachteil, dass diese keine Auskunft über die Kardinalitäten der Zwischenergebnisse geben. Wenn beispielsweise ein Graphmuster aus drei Tripelmustern t_1, t_2 und t_3 besteht, erhält man einen Anhaltspunkt über die Kardinalität von $t_1 \bowtie t_2 \bowtie t_3$ aber nicht über $t_1 \bowtie t_2$ oder $t_2 \bowtie t_3$. Da aufgrund des Speichermodells die Lösungsabbildungen von sternenförmigen Graphmustern ohne Join-Operationen berechnet werden können, wirkt sich dieser Nachteil nicht stark aus.

5.7 Verwandte Arbeiten

In diesem Abschnitt werden Ansätze aus der Literatur betrachtet, die sich mit der Optimierung von Anfragen in RDF-Datenbanksystemen beschäftigen. Der Fokus liegt dabei auf nicht-verteilten Datenbanksystemen, die ihre Daten auf dem Sekundärspeicher verwalten. Im Folgenden werden die verwandten Arbeiten zu den unterschiedlichen Teilen der Anfrageoptimierung beschrieben. Nacheinander wird auf die Themen Anfragemodelle für SPARQL, algebraische Transformationsregeln, Enumerierungsalgorithmen für die Generierung von Ausführungsplänen und Optimierungsstrategien in RDF-Datenbanksystemen eingegangen.

5.7.1 Anfragemodelle

In [NW10] wird eine SPARQL-Anfrage in ein internes Graphmodell überführt. Jeder Knoten in diesem Modell repräsentiert initial einen Scan über der Datenbank mit entsprechend gebundenen Variablen. Darüber hinaus werden Literale enthaltene Filterausdrücke berücksichtigt. Mit diesem Modell konzentrieren sich die Autoren auf die Auswerten von Join-Operationen zwischen Tripelmustern, d.h. Wahl der Join-Reihenfolge und der Ausführungsstrategien. Diese werden während der Generation des Anfragemodells festgelegt bzw. ausgewählt. Das SPARQL Query Graph Model hingegen betrachtet nicht einzelne Tripelmuster, sondern fokussiert die Optimierung bezüglich von sternförmigen Graphmustern.

5.7.2 Algebraische Transformationen

Algebraische Transformationsregeln werden auf SPARQL-Anfragen angewendet, um diese in semantisch äquivalente Anfragen umschreiben zu können. In [PAG09] beweisen die Autoren grundlegende semantische Äquivalenzen zwischen algebraischen SPARQL-Ausdrücken. Dazu gehören beispielsweise die Assoziativität, Kommutativität und Distributivität von Join und Vereinigungsoperatoren. Darüber hinaus zeigen sie, dass jede Anfrage in eine Normalform bestehend aus der Vereinigung von vereinigungsfreien Teilanfragen zerlegt werden kann.

In [V+10] befassen sich Vidal et al. mit der Ausführung von Anfragen, die sternförmige Graphmuster beinhalten. Hierbei führen sie die beiden Join-Operationen *njoin* und *gjoin* ein. Beim *njoin* wird der Join zwischen zwei Graphmustern über das Berechnen und Verknüpfen der Lösungen einzelner Tripelmuster ausgewertet. Im Falle eines *gjoin* werden zuerst die Lösungen für die sternförmigen Graphmuster berechnet und anschließend miteinander verknüpft. Für diese beiden Operatoren definieren die Autoren Transformationsregeln, die analog zu denen in [PAG09] definiert sind. Obwohl die Transformationsregeln in algebraischer Schreibweise präsentiert werden, entsprechen sie eher der Wahl einer geeigneten Auswertungsstrategie und bieten in Hinblick auf das algebraische Umschreiben von Anfragen keine neuen Erkenntnisse.

Aufbauend auf diese Erkenntnisse wurden in [Sch10] weitere Äquivalenzregeln nachgewiesen. Neben den in [PAG09] vorgestellten Regeln befassen sich die Autoren auch mit Idempotenzen und der Verschiebung und der Dekomposition von Projektion und Filterausdrücken.

In dieser Arbeit werden Äquivalenzregeln genutzt, um das SQGM einer Anfrage so umzuformen, dass die Vorteile des Resource Centered Store ausgenutzt werden können.

5.7.3 Enumerierungsalgorithmen

In [NW10] stellen die Autoren fest, dass eine Enumeration aller möglichen Ausführungspläne nicht möglich ist. In ihrem System bevorzugen sie Sort-Merge-Joins gegenüber Hash-Joins. Da stichproben-basierte und randomisierte Algorithmen sehr wahrscheinlich keine ordnungserhaltene Ausführungspläne für Anfragen mit mehr als 10 Joins generieren, verwenden sie dynamische Programmierung als Enumerierungsalgorithmus, wobei das Beschränken des Suchraumes über geschätzte Ausführungskosten erfolgt. Da in dem System einzelne Tripelmuster miteinander verknüpft werden, müssen bei der Generierung von Ausführungsplänen besonders auf die Erhaltung der Join-Reihenfolge geachtet werden (z. B. um sternförmige Graphmuster effizient zu verarbeiten). Obwohl beide Systeme dynamische Optimierung verwenden, werden im RCS sternförmige Graphmuster bereits bei der Generierung des initialen Ausführungsplans berücksichtigt und als ein einzelner Operator im Anfragemodell repräsentiert. Daher vereinfacht sich der Enumerierungsalgorithmus und somit spielen Strategien zur Ordnungserhaltung eine untergeordnete Rolle.

Dagegen wird in [V+10] ein randomizierter Algorithmus (random walk) für das Enumerieren der möglichen Ausführungspläne. Zu Beginn werden zufällige Ausführungspläne generiert. Anschließend werden Transformations-

regeln in Abhängigkeit einer Wahrscheinlichkeitsfunktion angewendet. Der Wahrscheinlichkeitswert hängt dabei von einem globalen über die Zeit hinweg variierenden Parameter. Dieser Parameter reflektiert die Anzahl der Optimierungsiterationen. Die angewendeten Regeln entsprechen im Wesentlichen den in Abschnitt 5.3.4 vorgestellten Transformationsregeln, wobei der Fokus ausschließlich auf Join-Operationen liegt und für den Join die zwei Ausführungsstrategien *njoin* und *gjoin* (vgl. Abschnitt 5.7.2) betrachtet werden.

Neben der dynamischen Programmierung wurden im Kontext von relationalen Datenbanksystemen auch heuristische Verfahren zur Optimierung des Ausführungsplans entwickelt, z. B. Randomisierte Algorithmen. Zu den randomisierten Algorithmen zählen beispielsweise Iterative Improvement, Simulated Annealing und Two Phase Optimization [IK90]. Randomisierte Algorithmen finden Anwendung, wenn die Anzahl der Relationen zu groß für dynamische Programmierung ist (mehr als 15-20). Neue Ausführungspläne werden bei diesem Herangehen durch eine Menge von Transformationsregeln generiert. Transformationsregeln werden so lange angewendet, bis ein lokales Minimum bzgl. Ausführungskosten erreicht worden ist. Um die Chancen für das finden eines optimalen Plans zu erhöhen, erlauben manche Algorithmen (z. B. Simulated Annealing) sogar ein zwischenzeitliches Erhöhen der Kosten. Im Allgemeinen funktionieren diese Verfahren am besten, wenn nur wenige lokale Minima existieren.

5.7.4 Optimierungsstrategien

Bei der Betrachtung der Optimierungsstrategien wird im Folgenden zwischen RDBMS-basierten und nativen RDF-Datenbanksystemen unterschieden.

5.7.4.1 RDBMS-basierte RDF-Datenbanksysteme

Ein auf relationaler Technologie basierendes RDF-Datenbanksystem generiert aus einer SPARQL-Anfrage eine SQL-Anfrage, um die Optimierungsstrategien der RDBMS wiederverwenden zu können. Zunächst wurden Regeln für die Überführung einer SPARQL-Anfrage in einen semantisch äquivalenten Ausdruck der relationalen Algebra betrachtet [Cyg05, HS05]. In den beiden Veröffentlichungen wurde die Notwendigkeit für SPARQL-spezifische Optimierungsstrategien andiskutiert. Beispielsweise wurde erkannt, dass durch das Eliminieren von temporären Tabellen oder Entschachteln von Unteranfragen ein besser zu optimierende Anfrage generiert wird.

Nach der Spezifikation von Regeln zum Generieren einer SQL-Anfrage aus einer SPARQL-Anfrage wurde in [CLF09] detaillierter eine Vor-Optimierung der generierten Anfrage diskutiert. Zu den darin genannten Vereinfachungen zählen beispielsweise das Eingrenzen der projizierten Variablen, das frühzeitige Projizieren von Variablen in Unteranfragen sowie das Vereinfachen von Join- und Vereinigungsoperationen bei Nichtvorhandensein von Left-Outer-Joins. In [CLAF08] beschrieben die Autoren einen speziellen, relationalen Operator, nested optional join, zum Berechnen von optionalen Graphmustern vor, der effizienter als das Ausführen eines Left-Outer-Join sein soll.

Die in diesem Bereich angewendeten Verfahren basieren ausschließlich auf Regeln und Heuristiken. Eine kostenbasierte Transformation der SPARQL-Anfragen an sich findet nicht statt.

5.7.4.2 Native RDF-Datenbanksysteme

Da sich relationale DBMS als Speichersystem für RDF-Daten nicht etablieren konnten, befasst sich der überwiegende Teil der Literatur mit der Optimierung von SPARQL-Anfragen in nativen RDF-Datenbanksystemen. Dieser Abschnitt beschreibt nacheinander verwandte Arbeiten über Heuristiken zur Anfragetransformation und Kostenabschätzungen.

Anfragetransformation Viele Ansätze zur Optimierung von SPARQL-Anfragen basieren auf Heuristiken, die Anfragen ohne das Einbeziehen von statistischen Informationen umschreiben. Zum einen wurden aus dem relationalem Bereich stammende Strategien auf RDF-Datenbanksysteme übertragen. Zum Beispiel werden möglichst kleine Zwischenergebnisse angestrebt, indem Filterausdrücke und Projektionen frühestmöglich ausgewertet werden [PPD+13, Jen10]. In Jena TDB werden darüber hinaus Filterausdrücke vor dem Verschieben im Anfragemodell in seine konjunktiven Teile zerlegt.

Zum anderen wird die Reihenfolge der Auswertung von Tripelmustern anhand der enthaltenen Konstanten festgelegt – je mehr Konstanten desto höher die Selektivität. In [SSB+08] wird angenommen, dass konstante Subjekte die höchste Selektivität haben, gefolgt von konstanten Objekten und Prädikaten. Diese Strategien werden in dem RDF-Repository Jena TDB angewendet. Tsialiamanis et al. legen eine Ordnung zwischen den Tripelmustern mit den verschiedenen Kombinationen von Konstante und Variablen in Tripelmuster fest [TSF+12]. Darüber hinaus bewerten die Autoren Literalen als selektiver als IRIs in Objektposition. Darauf aufbauend werden dann Präzedenzen von Joins zwischen Tripelmustern definiert. Diesen Optimierungsstrategien liegt die Annahme zugrunde, dass eine Anfrage Tripelmuster für Tripelmuster ausgewertet wird. Im Resource Centered Store werden ähnliche Strategien verwendet. Die Heuristiken im RCS berücksichtigen jedoch insbesondere sternförmige Muster und bewerten ein gegebenes Subjekt höher. Darüber hinaus werden in dieser Arbeit quasi-konstante Variablen eingeführt, für die spezielle Ausführungsstrategien zur Verfügung stehen.

Neben der Selektivität bevorzugen Ansätze wie [NW10] und [V+10] buschige Join-Bäume, da mit diesen Anfragen mit sternförmigen Graphmustern effizienter ausgeführt werden können. In [PPD+13] wird darüber hinaus angestrebt, sortierte Eingaben für Join-Operationen zu erzielen, um einen Sort-Merge-Join anstelle eines Hash-Joins verwenden zu können. Weiterhin können auf sortierten Mengen Duplikate (`DISTINCT`) einfacher eliminiert werden. Schließlich wird auch versucht, die Auswertung von Filterausdrücken mit Join-Operationen zu kombinieren.

In [LG13] beschreiben Loizou und Groth, wie Nutzer durch geschicktes Formulieren von SPARQL-Anfragen das Datenbanksystem zu dessen effizienten Ausführung beitragen können. Hierzu spezifizieren sie die folgenden fünf Heuristiken: (i) Eliminieren von optionalen Graphmustern aufgrund von statistischen Informationen, (ii) Verwenden von benannten Graphen zum Lokalisieren von Graphmustern, (iii) Verwenden von Aggregationsfunktionen zum Verkleinern von Zwischenergebnissen bei kartesischen Produkten und (iv) Ersetzen von Pfaden an Tripelmustern durch Pfadsequenzen zum Eliminieren von Variablen, (v) Ausnutzen von semantisch äquivalenten Notationen für das

Angeben von alternativen IRIs. Diese Heuristiken stellen eine Ergänzung zu den algebraischen Transformationsregeln dar.

Selektivität Neben Transformationsregeln werden in RDF-Datenbanksystemen auch statistische Informationen gespeichert und bei der Ermittlung des optimalen Ausführungsplans miteinbezogen. Für das Abschätzen der Kosten wird die Unabhängigkeit der Attributwerte (*engl. attribute value independence*) [PI97] als die größte Herausforderung angesehen. [SSB+08]

In ihrem System HPRD [BB07] verwenden Baolin et al. einfache Statistiken, um einen geeigneten Index für die Anfrageevaluation auszuwählen. Dabei werden für die Tripelmuster $(s, ?, ?)$, $(s, p_i, ?)$ und (s, p_i, o_k) jeweils die Anzahl der in der Datenbank passenden Tripel gespeichert. Die so gesammelten statistischen Informationen sind sehr umfangreich und können nur mit erhöhtem Aufwand aktuell gehalten werden. Die Autoren gehen auf diesen Fakt nur insofern ein, als dass sie ein Aktualisieren der statistischen Daten nur nach umfangreicheren Ladeoperationen („batch data load processing") und im Leerlauf des Systems vorsehen. Wie genau die Daten zur Anfrageoptimierung verwendet werden, wurde nicht näher ausgeführt.

Für das RDF-Datenbanksystem Jena2 [SSB+08, Jen10] werden je nach der gebundenen Komponente eines Tripelmusters unterschiedliche statistische Informationen genutzt. Im Falle eines gegebenen Subjekts wird die durchschnittliche Anzahl an Tripeln über alle Subjekte verwendet. Bei einem gegebenen Prädikat wird auf die Anzahl der Tripel mit diesem Prädikat und bei einem gegebenen Objekt auf ein Histogramm mit konstanter Klassenbreite (*engl. equi-width histogram*) zurückgegriffen. Darüber hinaus wird für jedes Prädikat die Verteilung der Objektwerte erhoben. Die Selektivität eines Tripelmusters ergibt sich als Produkt der Selektivitäten der einzelnen Komponenten und nimmt somit die statistische Unabhängigkeit der Komponenten an. Für die Berechnung der Selektivität des Joins zwischen zwei Tripelmustern werden Beziehungen zwischen Prädikaten, gegeben durch deren Definitions- und Wertebereich (`rdfs:domain` und `rdfs:range`), genutzt, um eine obere Schranke zu bestimmen. Das Betrachten von Definitions- und Wertebereich gehört zu dem Forschungsgebiet der semantischen Optimierung und wird in dieser Arbeit nicht näher betrachtet.

Im RDF-3X-System werden zwei Arten von Statistiken verwendet. Zum einen werden Statistiken zum Abschätzen der Selektivität von beliebigen Tripelmustern und Join-Operationen erhoben. Dabei werden Daten für jede mögliche Kombination von gebundenen Komponenten in einem Tripelmuster sowie für die möglichen Join-Partner von Subjekt, Prädikat und Objekt als Histogramm gespeichert. Zum anderen werden auch häufig auftretenden Pfaden betrachtet, diese beinhalten sowohl sternförmige Graphmuster als auch Pfade. Die häufigsten Pfade sowie deren Statistiken werden in einem Vorverarbeitungsschritt ermittelt. Zum Abschätzen der Kosten eine Anfrage wird diese in häufigste Pfade zerlegt und deren Selektivitäten miteinander multipliziert. Die häufigsten Pfade liefern jedoch nur eine Kardinalitätsabschätzung für das gesamte überdeckte Graphmuster aber nicht für die dazu zu berechnenden Join-Operationen. Obwohl der RCS auf diesem Ansatz basiert, wirkt sich dieses Problem aufgrund des verwendeten Speichermodells nicht so stark auf die

Anfrageoptimierung aus, da die Lösungsabbildungen für ein Sternmuster ohne Join-Operation berechnet werden können.

In [MASS08] werden Graph-Mining-Techniken, Maximum-Dependency-Tree und Pattern-Tree, zum Erheben von Statistiken über einem RDF-Graphen verwendet. Um die Selektivität eines Graphmusters zu bestimmen, werden darin enthaltene maximale Teilgraphen ermittelt, zu denen statistische Informationen existieren. Der Aufwand zum Erheben der statistischen Daten beträgt selbst für kleinere Datenmengen (ca. 10^6 Ressourcen) mehrere Stunden, wodurch dieser Ansatz nur eingeschränkt einsetzbar ist.

Mit der Verwendung von stichproben-basierten Verfahren wird in [V+10] ein anderer Weg eingeschlagen. Die Autoren greifen auf ein generisches Verfahren [LN90] zurück, mit dem auf Basis einer Grundmenge (*engl. population*) die Ergebnisgröße eine Anfrage berechnet werden kann. Die Abschätzungen werden basierend auf einem Urnenmodell getroffen. Die Vorteile sind bei diesem Ansatzes, dass keine Annahmen über die Charakteristiken der Daten getroffen werden und nur wenige Ressourcen für das Verwalten der statistischen Informationen verbraucht werden. Dem gegenüber steht die Problematik des Findens einer geeigneten Grundmenge und der geringere Präzision bei der Abschätzung der Selektivität.

In [HL11] werden Bayessche Netzwerke verwendet, um näherungsweise die Wahrscheinlichkeit des gleichzeitigen Auftretens von Ressourcen bzw. Literalen in einem Sternmuster voraussagen zu können. Dazu wird für häufige sternförmige Teilgraphen ein Bayessche Netzwerk gelernt. Die Komplexität des Lernalgorithmus ist dabei $O(n^4)$, wobei n die Anzahl der involvierten Eigenschaften ist. Des Weiteren muss bei diesem Vorgehen auf den Hauptspeicherverbrauch geachtet werden. Neben dem Bayesschen Netzwerk werden auch Histogramme über häufige Join-Pfade generiert. Mit Hilfe dieser Informationen erreichen die Autoren für Sternmuster eine höhere Genauigkeit bei der Abschätzung als [NW10].

Der überwiegende Teil der nativen RDF-Datenbanksysteme berechnen die Kosten für die Berechnung der Lösungen eines Graphmusters auf Basis der darin enthaltenen Tripelmuster, da die zugrunde liegenden Speichermodelle einzelne Tripel verwalten. Im Gegensatz dazu folgt der Resource Centered Store dem datenseiten-basierten Speichermodell von RDBMS, weshalb die Kosten für das Laden der relevanten Datenseiten wesentlich zu den Gesamtkosten für die Evaluation einer Anfrage beitragen. Derartige Kosten müssen von keinem der oben genannten Systeme berücksichtigt werden.

5.8 Zusammenfassung

In diesem Kapitel wurde die Anfragebearbeitung im Resource Centered Store betrachtet. Als systeminternes Modell zur Repräsentation von Operatoren und Datenflüsse wurde das SPARQL Query Graph Model (SQGM) entwickelt, das alle Phasen der Anfrageoptimierung unterstützt. Basierend auf existierenden Arbeiten wurden für das SQGM Regeln zur semantisch äquivalenten Transformation von Anfragen definiert. Im Anschluss daran wurden Ausführungsstrategien für die wichtigsten Operatoren der SPARQL-Algebra beschrieben. Insbesondere wurden neuartige Strategien für bestimmte Operatorkonstellationen entwickelt, die die Charakteristika des RCS-Speichermodells berück-

sichtigen. Aufgrund der Verwaltung der Tripel auf Datenseite konnte dabei eine Brücke zu den in relationalen DBMS verwendeten Join-Algorithmen geschlagen werden und erlaubt eine Wiederverwendung von vorhandenen Forschungsergebnissen.

Im Anschluss daran wurde die Generierung von Ausführungsplänen und die kostenbasierte Auswahl des auszuführenden Plans behandelt. Hierzu wurden speziell für das RCS ausgerichtete Heuristiken definiert, um für eine Anfrage eine gute initialen Ausführungsplan zu generieren. Insbesondere wurde hierbei der Zugriff auf die im vorherigen Kapitel beschriebene Indexe diskutiert. Die Bewertung eines Ausführungsplans erfolgt anhand einer Kostenfunktion, die auf statistische Informationen über den betrachteten RDF-Graph basiert.

Kapitel 6

Evaluation

In den vorangegangen Kapiteln wurde für die Komponenten Speicherung, Indexierung und Anfragebearbeitung jeweils alternative Strategien beschrieben. Für die Speicherung der RDF-Daten wurde ein Ansatz vorgeschlagen, der die Tripel eines RDF-Graphen bezüglich des Subjekts gruppiert auf dem Sekundärspeicher ablegt. Wie im Abschnitt 3.5 diskutiert wurde, werden Pfadanfragen nicht gut unterstützt, weshalb auf graphmuster-basierende Indexe entwickelt wurden, mit denen gezielt bestimmte Graphmuster (z. B. Pfade) indexiert werden können.

Dieses Kapitel beschäftigt sich mit der experimentellen Analyse der vorgeschlagenen Ansätze. Das Hauptziel der Evaluation ist zu zeigen, dass das in der Arbeit entwickelte Speichermodell für das Speichern und Anfragen von RDF-Daten geeignet ist. Darüber hinaus werden dessen Auswirkungen auf die Ausführungszeit bei der Beantwortung von Anfragen unterschiedlichen Characters hat. Zunächst werden in Abschnitt 6.1 grundlegenden Überlegungen zur Implementierung des Resource Centered Stores beschrieben und Design-Entscheidungen diskutiert.

Abschnitt 6.2 gibt einen Überblick über existierende RDF-Benchmarks, die zum Generieren von RDF-Datensätzen und als Anhaltspunkt für die Evaluation des RCS dienen können. Die beiden anschließenden Abschnitte 6.3 und 6.4 befassen sich mit der Evaluation des RCS bzw. der graphmuster-basierten Indexierung. In diesen Abschnitten wird auch auf die jeweiligen Ziele der Experimente und die Messgrößen eingegangen. Schließlich werden in Abschnitt 6.5 die in den Experimenten gewonnenen Erkenntnisse zusammengefasst.

6.1 Grundlegende Überlegungen

Die Implementierung des Speichermodells, der Indexierungsstrategie sowie der Anfrageverarbeitung basiert auf einem existierenden RDF-Datenbankmanagementsystem, das gleichzeitig als Vergleichssystem für die Evaluation von den genannten Komponenten dient. Dieser Abschnitt gibt zunächst einen Überblick über die grundlegenden Überlegungen zur Wahl des RDF-DBMS. Anschließend wird die Integration der jeweiligen Komponenten in die Architektur des gewählten RDF-DBMS erläutert und dann wesentliche Design-Entscheidungen beschrieben.

6.1.1 Datenbank-Framework

Der Fokus dieser Arbeit liegt nicht auf der vollständigen Implementierung eines RDF-DBMS, sondern vielmehr in der Evaluation der konzipierten Ansätze für Speicherung und Anfragen von RDF-Daten. Für die Implementierung dieser Ansätze wurde daher zunächst nach einem RDF-DBMS gesucht (im Folgenden als Framework bezeichnet), das zum einen grundlegende Funktionalitäten wie beispielsweise dem Parsen von SPARQL-Anfragen zur Verfügung stellt und zum anderen ein Verändern bzw. Erweitern der Datenhaltung und Anfragebearbeitung erlaubt. Abgesehen davon, dass natürlich die Quellen des Frameworks verfügbar sein müssen, fasst die folgende Liste die Anforderungen an das Framework zusammen:

Vollständiges RDF-DBMS. Das Framework ist vollständig bezüglich der Speicherung, Indexierung und Anfrageverarbeitung und kann daher als Referenzsystem für die Evaluation der in der Arbeit entwickelten Ansätze dienen.

Alternatives Speichermodell. Das Framework abstrahiert von dem zugrunde liegenden Speichermodell und gestattet die Integration von Speichermodellen von Drittanbietern.

Erweiterbare Anfragekomponente. Die Komponente für die Verarbeitung von Anfragen sollte die Möglichkeit bieten, neue Operatoren in die Anfrageausführung mit einzubeziehen.

Erweiterbarer Optimierer. Analog zu einer erweiterbaren Anfragekomponente muss sich auch der Optimierer anpassen lassen, um die Eigenschaften eines neuen Speichersystems einbeziehen zu können.

Ausgehend von der obigen Liste konnten zwei Frameworks identifiziert werden, die die Anforderungen erfüllen: Jena2 in Kombination mit dem Anfragesystem ARQ [WSKR03, Jen10] und Sesame [BKvH02]. Beide Systeme werden derzeit aktiv weiterentwickelt (Programmiersprache ist Java) und haben inzwischen den Status eines produktiv einsetzbaren RDF-DBMS erreicht. Sie unterstützen verschiedene Speichermodelle und können RDF-Daten im Hauptspeicher, in einer relationalen Datenbank sowie in einem auf B-Bäumen basierenden nativen Speichermodell verwalten.

Zur Einordnung der beiden Systeme werden im Folgenden die Ergebnisse des Berlin SPARQL Benchmarks (BSBM) aus den Jahren 2009 [BS09] und 2011 [BS11] betrachtet. Dazu ist anzumerken, dass das native Speichermodell von Sesame nur im ersteren gemessen wurde; im späteren wurde Sesame nur als Framework für BigOwlim genutzt. Zusammenfassend lässt sich feststellen, dass beide Systeme eher für kleinere Datenmengen (bis ca. 100 Mio. Tripel) geeignet sind.

Bei der Evaluation der in der Arbeit definierten Ansätze ist das Kriterium „Performanz bei großen Datenmengen" weniger relevant als eine gute Integration der Ansätze in ein RDF-Framework. Anstatt die Ansätze mit allen gängigen RDF-DBMS zu vergleichen, wird das für die Implementierung verwendete Framework als Referenzsystem verwendet.

Aufgrund ihrer Schichtenarchitektur wären beide System grundsätzlich gleich gut geeignet, jedoch hat sich bei der Analyse der Systemarchitektur gezeigt, dass die Programmierschnittstelle (API) von Jena2 klarer strukturiert ist und sich die neu zu entwickelnden Komponenten des RCS in Jena2 einfacher integrieren lassen. Aus den genannten Gründen wurde Jena2 als Framework für die Implementierung ausgewählt. Im Jena Framework sind zwei Speichermodelle implementiert: Jena TDB und Jena SDB. Als Referenzsystem wurde Jena TDB anstelle von Jena SDB gewählt, da es ebenfalls ein natives Speichermodell und keine relationale Datenbank verwendet.

6.1.2 Integration der Komponenten in Jena2/ARQ

Im Folgenden wird ein Überblick darüber gegeben, wie die in dieser Arbeit entwickelten Komponenten in die Jena2-Systemarchitektur integriert wurden. Abbildung 6.1 stellt die wesentlichen Komponenten der Systemarchitektur von Jena2 dar. Wie zu erkennen ist, folgt diese der grundlegenden Architektur von Datenbanksystemen, wie sie im Kapitel 1 beschrieben worden ist. Anhand dieser Abbildung wird zunächst der grundsätzliche Prozess der Anfragebearbeitung in Jena2 beschrieben. Anschließend wird für jede einzelne Schicht dargestellt, welche Komponenten für die Einbettung des Resource Centered Store in Jena2 realisiert worden sind.

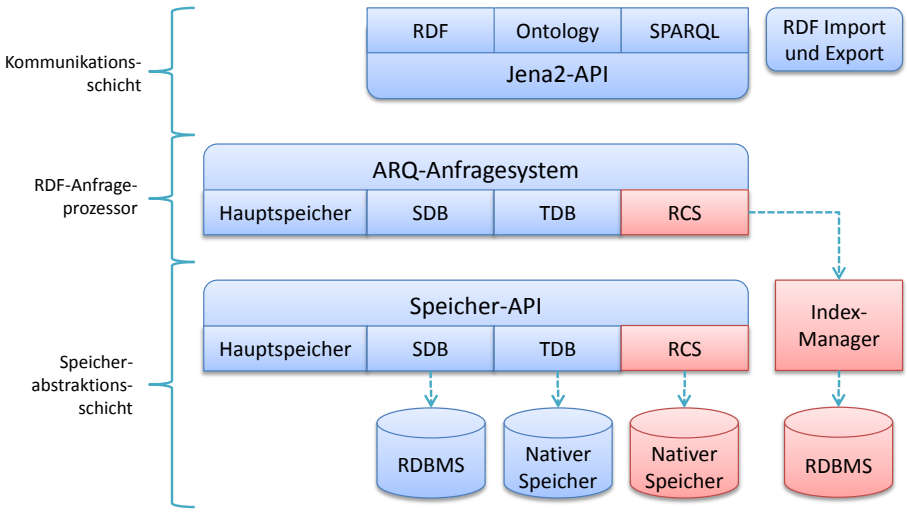


Abbildung 6.1: Systemarchitektur von Jena/ARQ mit den integrierten RCS-Komponenten

6.1.2.1 Anfragebearbeitung

Bevor eine Anwendung RDF-Daten in Jena2 speichern und anfragen kann, muss das Speichermodell festgelegt werden. Anschließend können über die *RDF-Import/Export*-Schnittstelle Daten in die Datenbank geladen werden, die dann entsprechend dem gewählten Speichermodell im Primär- oder Sekundärspeicher abgelegt werden.

Über die *Jena-API* kann die Anwendung dann auf die RDF-Daten zugreifen und beispielsweise SPARQL-Anfragen an das ARQ-Anfragesystem übergeben. Da die Jena-API für die Arbeit von geringerer Bedeutung ist, wird nicht näher darauf eingegangen.

Das *ARQ-Anfragesystem* ist die Komponente, die für die Verarbeitung und Optimierung von Anfragen verantwortlich ist. Eine Anfrage wird zunächst geparst und in ein internes Anfragemodell überführt, das auf der SPARQL-Algebra basiert. Dieses Modell wird anschließend transformiert, so dass ein optimalerer Anfrageausführungsplan (*engl. query execution plan – QEP*) entsteht. Da die Kosten für einen Anfrageausführungsplan von den Eigenschaften des zugrunde liegenden Speichermodells beeinflusst werden, werden entsprechend dem Speichermodells unterschiedliche Optimierungsstrategien verfolgt. Beispielsweise werden bei der statischen Optimierung in Jena TDB die Filterausdrücke unmittelbar nach dem zugehörigen Basis-Graphmuster ausgewertet.

Am Ende des Optimierungsprozesses entsteht ein QEP, dessen Operatoren während der Ausführung des Plans über die *Speicher-API* auf die RDF-Daten zugreifen. Dabei können auch speichermodellspezifische Zugriffsfunktionen genutzt werden.

6.1.2.2 Erweiterung von Jena2

Die Speicherabstraktionsschicht von Jena2 erlaubt das Realisieren von unterschiedlichen Speichermodellen. In Jena2 sind bereits drei Speichermodelle implementiert: Hauptspeicher, RDBMS und nativ. Diese Modellen können von der darüber liegenden Schicht über dieselbe Programmierschnittstelle zugegriffen werden. Aus dem zuvor beschriebenen Ablauf der Anfragebearbeitung ist ersichtlich, dass an zwei Stellen Informationen über das zugrunde liegende Speichermodell verwendet werden: Optimierer und QEP-Operatoren. Letztere erfordern im Allgemeinen auch spezialisierte Zugriffsfunktionen in der Speicher-API. Dementsprechend wurde für die Evaluierung des RCS dessen natives Speichermodell in die Speicher-API integriert. Darüber hinaus wurde auch ein auf RCS spezialisierter Optmimierer implementiert, der die Eigenschaften des Speichermodells berücksichtigt. Somit steht Anwendungen der RCS als Alternative zu den existierenden Jena2-Speichermodellen zur Verfügung.

Für die Evaluierung der graphmuster-basierten Indexierung wurde darüber hinaus der Optimierer so erweitert, dass dieser für geeignete Teile einer Anfrage den Zugriff auf Indexeinträge ermöglicht.

6.1.3 Realisierung des Resource Centered Store

Der Resource Centered Store wurde als eine Ergänzung zu den bereits existierenden Speichermodellen von Jena2 implementiert. In Abbildung 6.2 sind die im Kontext der Arbeit implementierten Komponenten dargestellt, wobei die grau hinterlegten Komponenten sind dem ARQ-Anfragesystem zuzuordnen sind. Realisierung des RCS lag der Fokus auf den Komponenten, die für eine Evaluation der vorgeschlagenen Ansätze entscheidend waren. Sofern es sinnvoll erschien, wurden existierende Systeme und Bibliotheken eingesetzt

(Datenbanksysteme und Bitset-Implementierung). In diesem Abschnitt werden zunächst die Komponenten für die Realisierung des Speichermodells näher betrachtet. Anschließend wird auf die Komponenten des Anfragesystems eingegangen.

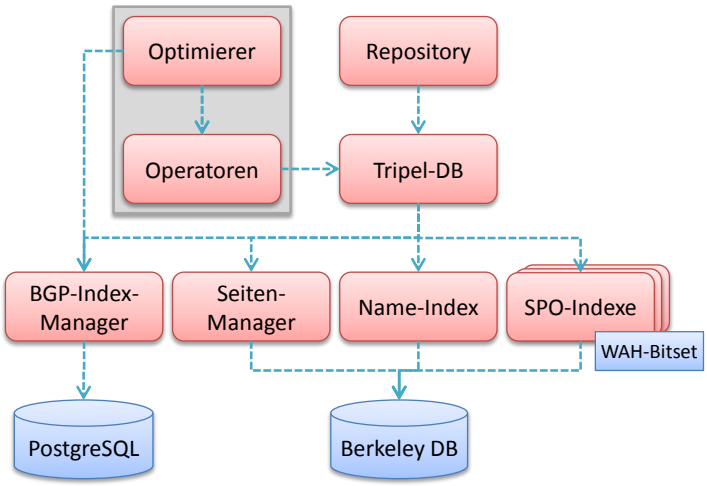


Abbildung 6.2: Wesentliche Komponenten des Resource Centered Stores

Die Implementierung des Speichermodells besteht aus den Komponenten Datenbanken, Indexmanager, Seitenmanager, Indexe, Tripeldatenbank und Repository. Im Folgenden werden der Reihenfolge nach die einzelnen Komponenten und ihr Zusammenwirken näher beschrieben.

6.1.3.1 Datenverwaltung

Für das Speichern der Daten wurden die existierenden Datenbanksysteme Berkeley-DB und PostgreSQL eingesetzt. In der *Berkeley-Datenbank* [Ber15] werden die RDF-Daten auf Datenbankseiten und deren Zugriffsstrukturen (Name-Index und SPO-Indexe) gespeichert. Für diese Aufgabe wurde eine Berkeley-DB gewählt, da diese auf die Verwaltung von Schlüssel-Wert-Paaren ausgelegt ist und entsprechende Caching-Algorithmen implementiert. Im Resource Centered Store werden Schlüssel-Wert-Paare an mehreren Stellen verwendet (vgl. Abschnitt 3.2.1). Beispielsweise werden die Datenbankseiten als Schlüssel-Wert-Paar behandelt, der Schlüssel ist dabei die Seiten-ID. Im Falle der Bitset-Indexe, die für das Lokalisieren von RDF-Termen benötigt werden, ist der ID eines RDF-Terms ein Bitarray zugeordnet.

Dahingegen wird für die Speicherung der graphmuster-basierten Indexe (BGP-Indexe) ein relationales DBMS verwendet, in der vorliegenden Implementierung wurde PostgreSQL [Pos13] eingesetzt. Für diesen Zweck ist ein relationales Datenbanksystem besser als Berkeley-DB geeignet, da ein Eintrag eines BGP-Indexes aus einer Menge von zusammengehörigen Variablenbindungen besteht (d.h., einer Variablen ist ein Wert zugeordnet), der sich geeigneter als Eintrag in einer Datenbankrelation darstellen lässt (vgl. Abschnitt 4.4.1). Darüber hinaus erlaubt eine relationale Datenbank das Filtern der Indexeinträge nach bestimmten Werten für eine Variable. Dementsprechend werden

alle Einträge für einen Index in einer separaten Relation in der Datenbank gehalten. Daneben beinhaltet die Datenbank auch noch ein Tabelle, in der Meta-Informationen über die vorhandenen Indexe gespeichert sind.

6.1.3.2 Abstraktionsschicht

Der *Seitenmanager* verwaltet die in einer Berkeley-DB gespeicherten Datenbankseiten und abstrahiert für die darüber liegenden Schichten von deren physischen Speicherung. Dieser hat keinerlei Informationen über den Inhalt der einzelnen Datenbankseiten, sondern realisiert nur einfache Funktionalitäten wie das Erstellen, Speichern und Löschen von Datenbankseiten und verwaltet die Seiten-IDs.

Wie auch der Seitenmanager bildet der *BGP-Indexmanager* eine Abstraktionsschicht, in diesem Falle wird von der physischen Speicherung der Indexe in der relationalen Datenbank abstrahiert. Der Indexmanager bietet ebenfalls nur eine einfache Schnittstelle zum Verwalten von Indexen. Mit diesem kann man manuell anhand eines Basis-Graphmusters einen Index angelegen und wieder löschen. Er bietet auch die Funktionalität zu einer gegebenen Anfrage alle zulässigen Indexe zu ermitteln, damit diese gegebenenfalls in den Anfrageausführungsplan integriert werden können.

6.1.3.3 Datenzugriffsstrukturen

Neben den BGP-Indexen existieren im RCS auch automatisch verwaltete Indexe, die dem effizienten Verwalten aller RDF-Daten dienen. Zu diesen hören zum einen der Name-Index und zum anderen die Subjekt-, Prädikat- und Objekt-Indexe (in Abbildung 6.2 als SPO-Indexe bezeichnet). Der *Name-Index* beinhaltet die 1:1-Abbildung zwischen einer ID und einer IRI bzw. einem Literal und wird als Schlüssel-Wert-Paar in der Berkeley-DB gespeichert. Neben dem automatisch von der Berkeley-DB generierten Primärindex auf dem ID-Wert, wurde auch ein sekundärer Index auf dem Wert (IRI oder Literal) angelegt. Dadurch werden beide Richtungen des Zugriffs, von der ID zum Namen und umgekehrt, effizient unterstützt.

Im Gegensatz zum Name-Index wird bei den *SPO-Indexen* nur eine Richtung des Zugriffs benötigt: von der ID zum Wert. Im Falle des Subjekt-Index besteht der Wert aus einer Zahl, die Seiten-ID und die Position des Subjekts auf der Seite kodiert. Im Gegensatz dazu besteht der Wert des Prädikat- und Objektindexes aus einem Bitset. Für die Bitset-Implementierung wurde auf eine existierende, angepasste Bibliothek zurückgegriffen, dem WAH-Bitset [WAH07].

6.1.3.4 Tripeldatenbank

Alle bisher beschriebenen Komponenten werden von der *Tripel-Datenbank* verwendet, um RDF-Daten auf dem Sekundärspeicher zu verwalten und für ein effizientes Lokalisieren zu indexieren. Mit einer Instanz einer Tripel-Datenbank werden nur die Tripel eines einzelnen (benannten) RDF-Graphen verwaltet. Die Tripel-Datenbank implementiert die für Speichermodell spezifischen Funktionen zum Hinzufügen und Löschen von Tripeln sowie das Berechnen von Lösungen für ein Basis-Graphmuster. Bei der Auswertung von sternförmigen Graphmustern kann beispielsweise die Information ausgenutzt werden,

dass für ein gegebenes Subjekt alle Tripel auf einer einzigen Datenbankseite abgelegt sind. Im Gegensatz zu der generischen Implementierung kann dieser auf die SPO-Indexe zugreifen und die relevanten Datenbankseiten bestimmen.

In einem *Repository* werden schließlich ein oder mehrere Tripeldatenbanken, der Default-Graph und Null oder mehr benannte Graphen) zusammengefasst. Die wesentliche Funktion des Repositories ist Anwendungen den Zugriff auf die Tripel-Datenbanken zu erlauben.

6.1.3.5 Anfrageverarbeitung

Damit das Anfragesystem die Eigenschaften des Resource Centered Store wie beispielsweise die Gruppierung der Tripel nach gemeinsamen Subjekten ausnutzen kann, wurden ein Optimierer sowie Operatoren implementiert. Der *Optimierer* transformiert anhand von Heuristiken das internen Anfragemodells, so dass ein effizienter Ausführungsplan entsteht. Beispielsweise werden Graphmuster zusammengefasst, die ein sternförmiges Graphmuster ergeben, und mit zugehörigen Filterausdrücken kombiniert, da hierfür ein auf den RCS spezialisierter Operator zur Verfügung steht. Der Optimierer greift auch auf den Index-Manager zu, um über die Verwendung von Indexen entscheiden zu können. Falls ein Index in einer Anfrage genutzt wird, generiert der Optimierer für das betreffende Graphmuster einen entsprechenden Operator.

Die *Operatoren* übernehmen während der Ausführung der Anfrage die Berechnung der Lösungen. Das ARQ-Anfragesystem beinhaltet generische Implementierungen für alle Operatoren der SPARQL-Algebra. Auch ohne spezialisierte Operatoren wäre die Ausführung von Anfragen über „unbekannte" Speichermodelle möglich, da diese die Speicher-API implementieren. Auf das jeweilige Speichermodell spezialisierte Operatoren haben jedoch den Vorteil, dass diese auf Funktionalitäten der Tripeldatenbank zugreifen können, die zum Berechnen der Lösungen eines Graphmusters auch Wissen über die Eigenschaften des Speichermodells ausnutzen. Darüber hinaus lassen sich alternative Algorithmen für das Ermitteln von (Teil-)Lösungen verwenden. Zum Beispiel wird der Zugriff auf einen BGP-Index über eine SQL-Anfrage an die PostgreSQL-Datenbank realisiert.

6.1.3.6 Versionsangaben der verwendeten Komponenten

Wie bereits erwähnt, wurden für die Implementierung existierende Software-Komponenten herangezogen. Tabelle 6.1 listet diese zusammen mit der jeweilig verwendeten Version auf. Die Bibliothek compressedbitset musste um einige Bitset-Operationen ergänzt, damit sie im RCS verwendet werden konnte.

Der Vollständigkeit halber ist in der Tabelle auch Jena TDB gelistet, die nicht in der Implementierung des RCS Verwendung findet, sondern nur als Vergleichssystem für die Evaluation herangezogen wurde.

6.2 Evaluation von RDF-Datenbanksystemen

Um letztendlich Benchmarks für das Bewerten des Leistungsverhaltens von RDF-Datenbanksystemen entwickeln zu können, wurden zunächst existierende RDF-Daten und später auch Anwendungsszenarien analysiert. In [MACP02]

Software	Version	Webseite
Apache Jena	2.6.4	http://jena.apache.org/
Apache Jena-ARQ	2.8.8	http://jena.apache.org/
Apache Jena-TDB	0.8.10	http://jena.apache.org/
Oracle Berkeley DB	3.3.75	http://www.oracle.com/technetwork/products/berkeleydb/
PostgreSQL	8.4	http://www.postgresql.org/
compressedbitset (modifiziert)	0.1	http://code.google.com/p/compressedbitset/

Tabelle 6.1: Verwendete Software-Komponenten

untersuchten beispielsweise Magkanaraki et al. 28 RDF-Schemata hinsichtlich der Anzahl von Klassen und Eigenschaften sowie den Umfang der Hierarchien. Anfänglich auf die Evaluation von RDF-Datenbanksystemen fokussiert [TCK05], erforschten Theoharis et al. die Eigenschaften von RDF-Daten und schufen einen Datengenerator für das Erzeugen von künstlichen Daten, die ähnliche Eigenschaften besitzen [TTKC08, TGC12]. Andere Forscher wählten für das Entwickeln eines Datengenerators ein konkretes Anwendungsszenario: Universitätsstrukturen [GPH05], Bibliographien (DBLP) [WGQH05, AMMH07, SHLP09] oder E-Commerce [BS09].

In diesem Abschnitt liegt der Fokus nicht auf RDF-Datengeneratoren an sich sondern auf dedizierte RDF-Benchmarks. Diese definieren neben einem Datengenerator auch eine Menge von Anfragen sowie eine Metrik für den Vergleich der Systeme. Im Folgenden werden die wichtigsten dedizierten RDF-Benchmarks besprochen, bei denen alle für die Durchführung einer Evaluation erforderlichen Werkzeuge, Daten und Informationen öffentlich zugänglich. Im Jahre 2005 wurde der erste dedizierte Benchmark für RDF-Datenbanksystem, der Lehigh University Benchmark, veröffentlicht [GPH05]. Einige Jahre später wurde das Thema in zwei Veröffentlichungen [SHLP09, BS09] wieder aufgegriffen. Diese Benchmarks werden im Folgenden näher beschrieben und abschließend gegenübergestellt.

6.2.1 Lehigh University Benchmark

Der erste verfügbar Benchmark wurde von Guo, Pan und Heflin unter dem Namen Lehigh University Benchmark (LUBM) [GPH05] entwickelt und basiert auf der Struktur einer Universität mit Entitäten wie beispielsweise Instituten, Vorlesungen und Studenten. Mit LUBM wollten die Autoren einen Benchmark schaffen, mit dem variable Datenmengen generiert und die Performanz von Anfragen evaluiert werden kann.

Das Schema der Daten ist nach Aussage der Autoren von moderater Größe und Komplexität. Neben einem Werkzeug für das Generieren der Daten beinhaltet der Benchmark auch 14 SPARQL-Anfragen, die verschiedene Arten von Anfragen abdecken sollen. Für die Wahl der Anfragen wurden Kriterien wie Anteil der involvierten Daten, Selektivität von Filterausdrücken und die Komplexität der Anfrage, wobei Komplexität über die Anzahl von verwendeten Klassen und Eigenschaften definiert ist.

Anfrage →	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Filterausdrücke, `OPTIONAL`, `UNION`, Modifikatoren nicht verwendet														
konst. Subjekt														
Sternmuster	1	2	1	1	1		1	2	2	1	1	2		
größtes Sternm.	2	3	2	2	5		2	3	3	2		2		

Tabelle 6.2: Übersicht über die Eigenschaften der Anfragen des LUBM

Darüber hinaus beinhaltet dieser Benchmark auch acht Anfragen, mit denen Inferenzkapazitäten (auf Basis von OWL) eines RDF-Datenbanksystem evaluiert werden (z. B. Typhierarchie von Klassen und Eigenschaften sowie die transitive und inverse Definition von Eigenschaften). Anhand der eher wenige Tripelmuster umfassenden Anfragen und der sehr umfangreichen Typhierarchie liegt die Vermutung nahe, dass der Benchmark im Wesentlichen auf die Evaluation der Inferenzfähigkeiten eines RDF-DBMS abzielt.

Tabelle 6.2 zeigt das Ergebnis einer Analyse der 14 Anfragen von LUBM. Man erkennt, dass der überwiegenden Teil (11 Anfragen) eine sternförmige Teilanfrage beinhaltet. Die sternförmigen Teilanfragen haben dabei meist eine Größe von zwei oder drei Tripelmustern, nur eine hat fünf Tripelmuster. Von diesen elf Anfragen bestehen sieben Anfragen aus einer einzelnen sternförmigen Teilanfrage, die restlichen haben zwei. Bei den letzteren sind die sternförmigen Teilanfragen direkt miteinander verknüpft, ein Objekt der einen Teilanfrage ist dabei das Subjekt der anderen. Bei zweien ist darüber hinaus ein Objekt der einen Teilanfrage gleichzeitig ein Objekt der anderen, d.h. die sternförmigen Teilanfragen sind über zwei Tripelmuster miteinander verbunden. Eine Verknüpfung der Anfragen nur über ein gemeinsames Objekt erfolgt nicht. Zwei der restlichen drei Anfragen ermitteln alle Ressourcen eines Typs (ein einzelnes Tripelmuster) und eine ist eine Pfadanfrage über zwei Tripelmuster.

Keine der LUBM-Anfragen beinhaltet Filterausdrücke oder weitergehende SPARQL-Sprachkonstrukte wie `OPTIONAL` oder `UNION`. Weiterhin ist bei den Anfragen auffällig, dass für alle Subjekte der sternförmigen Teilanfragen der Typ (`rdf:type`) angegeben ist.

In dem Benchmark wird eine Metrik mit drei Messgrößen definiert: Ladezeit, Speicherplatzbedarf der Datenbank, Ausführungszeit pro Anfrage sowie Vollständigkeit des Anfrageergebnisses. Letztere Metrik ist vor allem ein Indikator, mit welcher Qualität die Inferenzanfrage ausgeführt werden.

6.2.2 SP^2Bench

Die Autoren des SP^2Bench [SHLP09] verfolgten das Ziel, einen allgemeingültigen Benchmark zu entwickeln, der an kein konkretes Anwendungsszenario gebundenen ist. Mit den im Benchmark definierten Anfragen sollten möglichst viele unterschiedliche Einsatzbereiche für RDF-Datenbanksysteme abgedeckt werden.

Die Grundlage für den Benchmark bildet eine Analyse der bibliographischen Daten der DBLP [Ley09]. Die daraus gewonnenen Erkenntnisse über die Struktur der DBLP-Daten wurden genutzt, um einen Datengenerator zu realisieren, mit dem man einen beliebig großen RDF-Graphen mit ähnlichen Cha-

Anfrage →	1	2	3	4	5	6	7	8	9	10	11	12
Filterausdruck			×	×	×	×	×	×				
Join durch Filter				×		×			×			
`OPTIONAL`		×				×	×					
`UNION`								×	×			
Modifikator		×									×	
konstantes Subjekt		×			×				×		×	×
Sternmuster	1	1	1	2	2	1	3	4	1			
größtes Sternmuster	3	10	2	3	3	3	2	2	2			

Tabelle 6.3: Übersicht über die Eigenschaften der Anfragen des SP^2Bench

rakteristiken erzeugen kann. Die im Benchmark definierten Anfragen decken nach Aussage der Autoren die wichtigsten Sprachkonstrukte von SPARQL ab und erlauben das Testen einer Vielzahl von Datenzugriffs- und Optimierungsstrategien. Im Gegensatz zum LUBM wird Inferenz im SP^2Bench nicht berücksichtigt.

Anhand der SPARQL-Sprachkonstrukte und zweier Optimierungsstrategien sind insgesamt zwölf Anfragen bzw. Anfragetypen in dem Benchmark definiert worden. Bei den Sprachkonstrukten unterscheiden die Autoren zwischen Operatoren (z. B. `FILTER`, `OPTIONAL`), Modifikatoren (z. B. `LIMIT`, `OFFSET`) und Datenzugriff (z. B. anonyme Ressourcen, Literale). Bei den Optimierungsstrategien wurden zwei Konstellationen ausgewählt, die im Allgemeinen von Optimierern in relationalen DBMS berücksichtigt werden: Filterausdrücke möglichst früh ausführen (*engl. FILTER pushing*) und das Wiederverwenden von Zwischenergebnissen.

Eine Analyse der zwölf Anfragen dieses Benchmarks hat gezeigt, dass diese ein breites Spektrum abdecken (vgl. Tabelle 6.3). Es reicht von einfachen Anfragen bestehend aus einem einzelnen Tripelmuster bis hin zu Anfragen, die mehrere optionale Graphmuster sowie Filterausdrücke beinhalten. Neun der zwölf Anfragen beinhalten ein oder mehr sternförmige Teilanfragen. Fast alle haben eine Größe von zwei oder drei Tripelmustern, nur eine besteht aus zehn Tripelmustern. Die Verknüpfung der sternförmigen Teilanfragen erfolgt auf mehreren Arten (in Klammern die Anzahl der Anfragen): Filterausdruck (4), Subjekt der einen Teilanfrage ist Objekt der anderen (2) und gemeinsames Objekte (2). Bei einer Verknüpfung über einen Filterausdruck werden neben Gleichheit auch andere Komperatoren wie kleiner als oder ungleich verwendet. Insgesamt sieben Anfragen verwenden einen Modifikator, wobei `DISTINCT` fünf, `ORDER BY` zwei und `LIMIT/OFFSET` ein Mal vorkommen. Anonyme Ressourcen werden in fünf Anfragen verwendet.

In ihrer Publikation [SHLP09] definieren Schmidt et al. darüber hinaus fünf Metriken, die nach Möglichkeit für jede Evaluation gemessen werden sollten: Erfolgsrate der Anfragen,[1] Ladezeit für die Daten, Ausführungszeit pro Anfrage, Ausführungszeit über alle Anfragen sowie Speicherplatzbedarf im Hauptspeicher und auf dem Sekundärspeicher.

[1] Es werden hierbei die folgenden Status unterschieden: Erfolg, Timeout, Überlauf des Hauptspeichers und allgemeiner Fehler.

Anfrage →	1	2	3	4	5	6	7	8	9	10	11	12
Filterausdruck	×		×	×			×	×	×	×		
Join durch Filter												
`OPTIONAL`		×	×				×	×				
`UNION`				×							×	
Modifikator	×		×	×	×			×		×		
konstantes Subjekt		×			×				×		×	×
Sternmuster	1	1	1	2	2	1	3	1		1		2
größtes Sternmuster	5	13	7	5	4	2	5	9		6		6

Tabelle 6.4: Übersicht über die Eigenschaften der Anfragen des BSBM

6.2.3 Berlin SPARQL Benchmark

Der von Bizer und Schultz entwickelte Berlin SPARQL Benchmark (BSBM) basiert auf einem E-Commerce-Anwendungsszenario, in dem mehrere Klienten eine über das Internet zugängliche RDF-Datenbank (*engl. SPARQL endpoint*) zugreifen. [BS09] Beim Entwurf dieses Benchmarks stand weniger die Performanz für einzelne Anfragen im Vordergrund, ebenso wenig werden die Inferenzkapazitäten der Systeme bewertet. Vielmehr wollten die Autoren einen Benchmark schaffen, in dem ein realistischer Workload für das jeweilige Datenbanksystem generiert wird und die Anfrageperformanz für große Datenmenge gemessen werden kann.

Das Anwendungsszenario ist ein Online-Shopsystem, auf das mehrere Nutzer (gleichzeitig) zugreifen. Für die Daten wurde einfaches Schema mit Produkten, Reviews, Hersteller und Verkäufer gewählt. Der Datengenerator erzeugt Instanzen dieser Entitäten in einem festgelegten Verhältnis zueinander. Aus der Veröffentlichung wird nicht ersichtlich, auf welcher Grundlage diese Verhältnisse gewählt wurden.

Im BSBM werden nicht nur eine Reihe von Anfragen sondern auch eine Anfragemischung spezifiziert. Die Anfragemischung simuliert die Suche von Kunden nach einem Produkt und deren anschließende Navigation durch das Shopsystem und besteht aus einer Sequenz von 25 Schritten. In jedem Schritt wird eine von den zwölf im BSBM definierten Anfragen ausgeführt. Diese Sequenz wird als Anfragemischung (*engl. query mix*) bezeichnet. Damit die Performanz der RDF-Datenbanksysteme mit denen von relationalen DBMS verglichen werden können, wurde auch eine SQL-Variante der Anfragemischung in dem BSBM definiert.

Tabelle 6.4 gibt einen Überblick über die wesentlichen Eigenschaften der Benchmark-Anfragen. Zehn der zwölf Anfragen basieren auf sternförmigen Graphmustern, wobei vier davon zwei bzw. drei Sternmuster beinhalten. Als Besonderheit wurden bei drei Anfrage die Subjekte der Sternmuster festgelegt, d.h. die das Subjekt dieser Muster ist eine Ressource und keine Variable. Obwohl sieben der zwölf Anfragen mindestens einen Filterausdruck enthalten, werden diese nicht als Join verwendet. Vielmehr handelt es sich bei diesen um Bereichsanfragen, es werden die Werte für eine Eigenschaft eingeschränkt.

Darüber hinaus werden in fünf Anfragen Modifikatoren, in vieren `OPTIONAL` und in zweien `UNION` eingesetzt. Schließlich definiert der Benchmark auch

eine DESCRIBE- und einen CONSTRUCT-Anfrage. Zu ersterer ist anzumerken, dass alle Tripel zu allen möglichen Objekten eines Tripelmusters beschrieben werden. Diese ist somit mit einer sternförmigen Anfrage vergleichbar, bei der die Werte für das Subjekt eingeschränkt sind.

In dem BSBM werden drei Metriken definiert, die bei der Durchführung eines Benchmark-Tests gemessen werden sollen: Anfragemischungen pro Stunde (*engl. Query Mixes per Hour – QMpH*), Anfragen pro Sekunde (*engl. Queries per Second – QpS*) und Ladezeit. Anhand der Metriken ist ersichtlich, dass Fokus dieses Benchmarks nicht auf der Ausführungszeit einer einzelnen Anfrage sondern auf dem Durchsatz liegt. Der von einem RDF-Datenbanksystem verbrauchte Speicherplatz wird in diesem Benchmark nicht berücksichtigt.

6.2.4 Zusammenfassung und Vergleich der Benchmarks

Im Allgemeinen lassen sich zwei Richtungen für das Entwickeln von Anfragen eines Benchmarks unterscheiden [Gra93]. Die Anfragen werden dahingehend entwickelt, dass zum einen bestimmte Eigenschaften einer Anfragesprache oder eines Datenmanagementsystem und zum anderen bestimmte Anforderungen eines realen Anwendungsszenarios getestet werden. Während LUBM und SP^2Bench zu der ersten Kategorie von Benchmarks zählen, gehört der BSBM eher zu der zweiten. Dennoch ist den zuvor beschriebenen Benchmarks gemeinsam, dass sich die Autoren an realen Datensätzen orientiert haben. Die unterschiedliche Ausrichtung spiegelt sich in der Spezifikation der Anfragen und der Metriken wider.

6.2.4.1 Komplexität der Anfragen

Die Anfragen des Lehigh University Benchmark bestehen aus Basis-Graphmustern ohne Filterausdrücke oder andere SPARQL-Sprachkonstrukte (vgl. Tabelle 6.2) und dienen dem Bewerten der Performanz von Inferenzanfragen. Die Komplexität der Anfrage wird hier anhand der Inferenz über dem zugrunde liegenden RDF-Schema definiert. Der Fokus dieses Benchmarks liegt vor allem auf der Spezifikation eines RDF-Schemas, das möglichst umfangreiche Inferenzen erlaubt.

Einen anderen Begriff der Komplexität wird im SP^2Bench verwendet. Bei diesem Benchmark liegt der Fokus auf der Definition von Anfragen, die alle möglichen Kombinationen von SPARQL-Sprachkonstrukte beinhalten. Die Anfragen wurden so gestaltet, so dass zum Berechnen von deren Lösungen rechenintensive Operationen erforderlich sind. Zum Beispiel finden sich in diesem Benchmark mehrere in einem Filterausdruck spezifizierte Join-Operationen zwischen Sternmustern und Anfragen mit geschachtelten OPTIONAL-Klauseln. Der Fokus dieses Benchmarks liegt somit auf der Ausführungszeit einzelner Anfragen.

Im Gegensatz zu den bisher erwähnten Zielen verfolgt der Berlin SPARQL Benchmark das Ziel den Durchsatz eines RDF-Datenbanksystems zu bewerten. Die Komplexität besteht hierbei in der Hintereinanderausführung von Anfragen, in der verschiedene SPARQL-Sprachkonstrukte verwendet werden. Zusätzlich wird die Last für das RDF-Datenbanksystem dadurch erhöht, dass mehrere Klienten gleichzeitig Anfragen an das System absetzen. Dieser Benchmark ist der einzige von den drei genannten, der weiterentwickelt und an die

neue SPARQL-Version angepasst wurde. Mit der Version 3 des BSBM können auch die Performanz von Update-Operationen bewertet werden. Im Vergleich mit den anderen beiden Benchmarks ist auffällig, dass nur der BSBM unterschiedliche Datentypen für Literale in den Benchmark mit einbezieht.

6.2.4.2 Metriken

In allen Benchmarks werden Metriken zum Vergleichen der Performanz der RDF-Datenbanksystem festgelegt. Die wesentlichen Merkmale sind dabei die benötigte Zeit für das Laden der RDF-Daten in die Datenbank und die Ausführungszeit der Anfragen. Beim BSBM wird die Ausführungszeit implizit durch die Metrik „Anzahl von Anfragen pro Stunde" ausgedrückt.

Während beim LUBM und SP^2Bench auch der benötigte Speicherplatz gemessen wird, wird diese Metrik beim BSBM nicht berücksichtigt. Ein weiterer Unterschied bei den Benchmarks ergibt sich bezüglich der Vollständigkeit des Anfrageergebnisses. Beim SP^2Bench und BSBM spielen implizit im Graph enthaltene Aussagen keine Rolle, weshalb die Ergebnisse – eine korrekte Implementierung der SPARQL-Semantik vorausgesetzt – auf jeden Fall vollständig sind. Im Falle des LUBM ist dahingegen Inferenz ein wesentlicher Aspekt des Benchmarks.

Aufgrund der Analyse der Benchmarks wurde für die Evaluation des RCS der Berlin SPARQL Benchmark ausgewählt. Der Lehigh Univeristy Benchmark fokussiert vor allem die Inferenzfähigkeiten eines RDF-DBMS; jedoch werden Inferenzen in dieser Arbeit nicht betrachtet. Von den anderen beiden Benchmarks, SP^2Bench und BSBM, wird nur der letztere noch weiterentwickelt. Darüber hinaus werden anhand dieses Benchmarks regelmäßige Evaluationen von den verbreitetsten RDF-DBMS durchgeführt, so dass bei einer Weiterentwicklung des RCS eher ein Vergleich mit anderen Systemen möglich ist.

6.3 Evaluation des Speichermodells

Die Evaluationskriterien der in Abschnitt 6.2 beschriebenen Benchmarks zielen auf das Messen der Ausführungszeiten von Anfragen und Ladeoperationen ab. Obwohl bei der Implementierung des Resource Centered Stores auch Leistungsaspekte berücksichtigt wurden, stellt jedoch das System eher ein Proof-of-Concept dar und es wurden Kompromisse eingegangen. Beispielsweise wird für das Persistieren von Datenbankseiten und Indexen die Berkeley-DB anstelle einer dedizierten Komponente verwendet. Dies legt die Hypothese nahe, dass die für das RCS gemessenen Zeiten nicht direkt mit denen von existierenden Systemen vergleichbar sind.

Darüber hinaus wurden in den jeweiligen Experimenten auch die Ausführungszeiten für die Anfragen gemessen, um das System in Relation zu existierenden System zu setzen und einen Eindruck von den Eigenschaften des Speichermodells zu schaffen. Der Fokus der in diesem Abschnitt beschriebenen Experimente liegt jedoch eher auf dem Untersuchen der Komplexität der Algorithmen und dem Vergleich mit den theoretischen Abschätzungen aus den Abschnitten 3.2.4 und 5.6.

Im Folgenden werden das Vorgehen und die Ergebnisse der Evaluation des Speichermodells des Resource Centered Stores (vgl. Kapitel 3 beschrieben. Im

Abschnitt 6.3.1 werden zunächst Details zu der Konfiguration der Experimente und deren Durchführung gegeben. Anschließend werden im Abschnitt 6.3.2 die Messergebnisse der Experimente präsentiert und Beobachtungen notiert. In Abschnitt 6.3.3 werden die Messergebnisse interpretiert und zum Speichermodell in Beziehung gesetzt. Daraus lassen sich dann Aussagen über die Qualität des Speichermodells treffen und mögliche Alternativen diskutieren.

6.3.1 Konfiguration und Durchführung

In diesem Abschnitt werden zunächst grundlegende Informationen zur Evaluation des RCS-Speichermodells geben. Zunächst werden die Eigenschaften der Anfragen beschrieben, die für die Evaluation verwendet wurden. Im Anschluss daran wird auf die für die Evaluation relevanten Messgrößen eingegangen und das Messverfahren erläutert. Zuletzt wird die Evaluationsumgebung wie Rechner- und Softwarekonfiguration präsentiert.

6.3.1.1 SPARQL-Anfragen

Die Analyse der verfügbaren RDF-Benchmarks hat gezeigt, dass diese auf eine ganzheitliche Evaluation der Performanz eines RDF-Datenbanksystems ausgerichtet sind. Daher konnten die in den Benchmarks enthaltenen Anfragen nur als Anhaltspunkt genommen werden. Die Hauptcharakteristik des RCS ist das Gruppieren der Tripel anhand der Subjekte und somit insbesondere die Unterstützung von sternförmigen Anfragen. Zum Untersuchen dieser Eigenschaft wurden speziell konstruierte Serien von Anfragen erstellt, deren Kern sternförmige Basis-Graphmuster bilden. Aus den Ergebnissen dieser Experimente können anschließend Aussagen über beliebige SPARQL-Anfragen geschlussfolgert werden. Diese Schlussfolgerungen sind aufgrund zweier Eigenschaften des RCS möglich. Zum einen stellen sternförmige Basis-Graphmuster die kleinste atomare Einheit einer SPARQL-Anfragen im RCS dar, d.h. eine beliebige SPARQL-Anfrage kann in eine Menge von sternförmigen Anfragen zerlegt werden. Zum anderen entspricht die Systemarchitektur des RCS weitestgehend der eines relationalen DBMS. Somit können die Erkenntnisse aus der Forschung zur Verarbeitung von SQL-Anfragen (z. B. Kostenabschätzungen für Join-Operationen) auf den RCS übertragen werden.

Sternförmige Anfragen. Um die Verarbeitung von sternförmigen Graphmuster im RCS zu evaluieren, wurden Anfragen der Form `SELECT * WHERE { <pattern> }` im RCS ausgeführt und alle Lösungsabbildungen gelesen. Dabei wurden die folgenden Eigenschaften des Basis-Graphmusters variiert:

I) *Variable/Konstante im Subjekt.* Aufgrund der Eigenschaften des Speichermodells können sternförmige Basis-Graphmuster mit gegebenen Subjekt mit konstanter Komplexität beantworten werden. Durch den Vergleich von Anfragen mit Konstante bzw. Variable in Subjektposition kann die theoretisch angenommene Komplexität validiert werden.

II) *Anzahl der Tripelmuster.* Durch Variation der Anzahl an Tripelmuster kann der Nutzen des Gruppierens der Tripel anhand des Subjekts bewertet werden. In der Arbeit wurde die Hypothese aufgestellt, dass mit

wachsender Anzahl an Tripelmuster ein größerer Leistungsvorteil zu bestehenden Speichermodellen erzielt wird.

III) *Anzahl der Konstanten.* Die Anzahl der im Basis-Graphmuster enthaltenen konstanten Werte (Ressourcen und Literale) bestimmt wesentlich, welchen Nutzen die Prädikat- und Objektindexe für das Bestimmen der für die Anfrage relevanten Datenseiten haben. Die Hypothese dabei ist, dass mit wachsender Anzahl von Konstanten der Nutzen der Indexe wächst, d.h. (fast) alle selektierten Datenseiten sind relevant.

IV) *Ergebnisgröße.* Da alle Lösungsabbildungen für eine Anfrage gelesen werden, gibt das Variieren der Ergebnisgröße einen Einblick, in den durch den Zugriff auf Datenseiten verursachten Aufwand. Es ist zu erwarten, dass mit wachsender Ergebnisgröße der Einfluss des Datenmanagements auf die die Gesamtausführungszeit steigt.

Join von sternförmigen Anfragen. Im Anschluss an das Betrachten von sternförmigen Anfragen wird untersucht, wie sich eine Join-Operation zwischen zwei sternförmigen Anfragemustern auf die Ausführungszeit auswirkt. Zum Evaluieren dieser Art von Ausführungsplänen wurden Anfragen bestehend aus zwei sternförmigen Graphmustern konstruiert, die über eine Variable oder eine Konstante miteinander verknüpft werden – ein Join über mehrere Komponenten wurde in dieser Arbeit nicht untersucht. Für das Verknüpfen zweier sternförmiger Graphmuster ergeben sich damit grundsätzlich die folgenden Varianten:

OV Eine Variable ist ein gemeinsames Objekt

OK eine Konstante ist ein gemeinsames Objekt,

SV eine Variable ist das Subjekt des einen und ein Objekt des anderen Sternmusters, und

SK eine Konstante ist das Subjekt des einen und ein Objekt des anderen Sternmusters.

Von diesen vier Fällen werden in der Evaluation nur die ersten drei betrachtet, da die letzte Form nur ein Spezialfall der Ausführung einer sternförmigen Anfrage ist. Da in diesem Fall das Subjekt eine Konstante des einen Sternmusters ist, können dessen Lösungen mit einer Komplexität von $O(1)$ berechnet werden und es verbliebe nur das Berechnen der Lösungen des anderen Sternmusters. Für die ersten drei Fälle wurden Gruppen von je sechs Anfragen in Hinblick auf die nachfolgend beschriebenen Eigenschaften konstruiert.

Die an der Join-Operation beteiligten Graphmuster beinhalten ein selektives Tripelmuster (vgl. Abbildung 6.3, Zeile 3 und 6), d.h. ein Zwischenergebnis enthält nicht alle Ressourcen eines RDF-Typs (z. B. alle Produkte). Tabelle 6.5 listet die Größen der Zwischenergebnisse und des Endergebnisses auf.

Die Anfragen einer Gruppe bestehen aus sechs Kombinationen von sternförmigen Teilmuster mit 2, 4, oder 6 Tripelmuster, wobei alle Tripelmuster aus einem konstanten Prädikat und einer Variable als Objekt bestehen (vgl. Abbildung 6.3, Zeile 7). Die evaluierten Kombinationen von rechtem und linkem

```
select * where {
        ?review
                dc:date "2008-06-18"^^xsd:date ;
                v:reviewFor ?product .
        ?product
                v:producer i:Producer1 ;
                rdf:label ?label ;
                dc:date ?date ;
                v:productPropertyNumeric1 ?n .
}
```

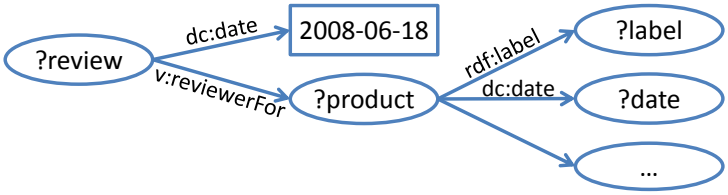


Abbildung 6.3: Beispielanfrage mit einer Join-Operation zwischen sternförmigen Graphmustern vom Typ SV (oben: SPARQL, unten: graphisch)

Join-Operand sind 2–2, 4–2, 6–2, 4–4, 6–4 und 6–6, wobei die Zahlen die jeweiligen Größen des linken und des rechten Sternmuster angeben. Vorbereitende Messungen haben gezeigt, dass die Ausführungszeit sich nicht signifikant verändern, wenn die Position der beiden Graphmuster im Ausführungsplan ausgetauscht werden.

6.3.1.2 Messgrößen und Messverfahren

Die Evaluation des RCS-Speichermodells basiert auf der experimentellen Untersuchung der Komplexität der Algorithmen zum Berechnen der Lösungsabbildungen von Anfragen. Im Gegensatz zu den vorgestellten Benchmarks ist die Ausführungszeit von sekundärem Interesse. Obwohl die Ausfürhungszeiten für die Anfragen gemessen wurden, war das Ziel des Prototyps nicht, *das* effizienteste RDF-DBMS zu realisieren, sondern eher das RCS-Speichermodells zu validieren. Daher liegt der Fokus eher auf dem Vergleich zwischen den Ausführungszeiten der verschiedenen Gruppen von Anfragen.

Join-	10k			100k			500k			1000k		
Art	$\|\Omega_r\|$	$\|\Omega_l\|$	$\|\Omega\|$	$\|\Omega_r\|$	$\|\Omega_l\|$	$\|\Omega\|$	$\|\Omega_r\|$	$\|\Omega_l\|$	$\|\Omega\|$	$\|\Omega_r\|$	$\|\Omega_l\|$	$\|\Omega\|$
OV	26	24	10	18	16	8	37	39	11	66	59	10
OK	7	7	49	7	7	49	7	7	49	7	7	49
SV	8	30	8	20	45	10	272	42	10	215	80	10

Tabelle 6.5: Ergebnisgröße des rechten und linken Graphmusters des Joins sowie Größe des Endergebnisses für die Join-Varianten OV, OK und SV

In Tabelle 6.6 sind die in den Experimenten erhobenen Messgrößen zusammengefasst.

Messgröße	Bedeutung
$\lvert p_r \rvert$	Anzahl der zugegriffenen Datenseite
$\lvert p_\Omega \rvert$	Anzahl der distinkten Datenseiten, die mindestens eine Lösungsabbildung enthalten
$\lvert I_s \rvert$	Anzahl der Zugriffe auf den Subjektindex
$\lvert I_p \rvert$	Anzahl der Zugriffe auf den Prädikatsindex
$\lvert I_o \rvert$	Anzahl der Zugriffe auf den Objektindex

Tabelle 6.6: Messgrößen zur Evaluation des RCS

Für jede Anfrage wird darüber hinaus die Ausführungszeit gemessen. Diese Beinhalten das Ausführen und das Abrufen aller generierten Lösungsabbildungen, jedoch nicht das Parsen der Anfrage oder das Generieren des Ausführungsplans. Bevor die erste Anfrage an das Datenbanksystem gesendet wurde, wurde zum Warmlaufen des Systems eine Anfrage ausgeführt. Damit Caching vernachlässigt werden kann, wurde jede Anfrage 10 mal hintereinander ausgeführt. Da die Zugriffsstatistiken auf die Datenseiten nur vom verwendeten Ausführungsplan abhängt, wurden diese durch ein zusätzliches Ausführen der jeweiligen Anfrage erhoben.

Damit die für das RCS gemessenen Ausführungszeiten zumindest qualitativ bewertet werden können, wurden alle Anfragen auf denselben Datensätzen mit dem Datenbanksystems Jena TDB [Jen10] ausgeführt. Jena TDB eignet sich als Vergleichssystem, da es bezüglich der Anfragegenerierung dieselben Software-Komponenten benutzt, aber ein tripelbasiertes, auf B-Bäumen basierendes Speichermodell verwendet. Es können somit der Einfluss von Software-Komponenten minimiert werden, die nicht direkt mit dem verwendeten Speichermodell zusammenhängen. Darüber hinaus ist ein direkter Vergleich mit einem auf Performanz ausgerichteten, alternativen Speichermodell möglich und erlaubt daher Rückschlüsse auf die Implementierung des RCS.

6.3.1.3 Evaluationsumgebung

Zur Generation der Testdaten wurde der Datengenerator des Berlin SPARQL-Benchmarks (BSBM) verwendet (vgl. Abschnitt 6.2.3). Der Datengenerator des BSBM wurde ausgewählt, da dieser sich auf die Auswertung von sternförmigen Anfragen fokussiert und eine dem RCS entgegen kommende Anwendungsumgebung (Online E-Commerce-System) beschreibt.

Die Experimente wurden auf einem Server unter dem Betriebssystem Solaris 10 (64 Bit). Der Server verfügt über vier CPUs vom Typ Dual-Core AMD Opteron und 32 GB Hauptspeicher. Zum Ausführen der Software wurde Java in der Version 1.7.0_21 verwendet. Die verwendeten Software-Komponenten sind zusammen mit der jeweilig verwendeten Version in Tabelle 6.1 auf Seite 140 aufgelistet.

6.3.2 Ergebnisse und Beobachtungen

In diesem Abschnitt werden die erhobenen Messergebnisse dargestellt und während der Experimente gemachten Beobachtungen notiert. Eine detaillierte Diskussion der Messwert und Beobachtung sowie deren Bedeutung für das RCS-Speichermodell erfolgt im anschließenden Abschnitt 6.3.3. Zunächst werden die in den Experimenten verwendeten Datensätze und ihre allgemeinen statistische Informationen beschrieben. Daran anschließend werden die Ergebnisse der Auswertung von Anfragen präsentiert.

6.3.2.1 Datensätze

Für die Evaluation des RCS-Speichermodells wurden insgesamt vier Datensätze unterschiedlicher Größe mittels des BSBM-Datengenerators [BSB13] generiert. Im folgenden werden diese als *10k*, *100k*, *500k* und *1000k* referenziert. In diesem Abschnitt werden Informationen zum Speicherplatzbedarf, zur Datenverteilung und zu den Ladezeiten gegeben.

Speicherplatzbedarf. Tabelle 6.7 stellt die Anzahl der Tripel für die vier Datensätze sowie den verwendeten Speicherplatz und die Anzahl der belegten Datenseiten dar. Für die Experimente wurde die Größe einer Datenseite mit 16 KB konfiguriert. Zum Vergleich wurde auch der Speicherplatzbedarf der entsprechenden TDB-Datenbanken angegeben.

Datensatz	Datei	$\lvert\mathcal{T}\rvert$	$\lvert\mathsf{p}\rvert$	$\lvert\mathcal{T}\rvert/\lvert\mathsf{p}\rvert$	RCS	TDB	TDB/RCS
10k	3 MB	11.962	15	798	18 MB	3 MB	6,0
100k	23 MB	97.795	84	1164	44 MB	20 MB	2,2
500k	124 MB	510.838	347	1147	226 MB	103 MB	2,2
1000k	261 MB	1.066.626	632	1688	458 MB	217 MB	2,1

Tabelle 6.7: Datensatzgrößen und Speicherplatzverbrauch

Beobachtung 6.1. Im Vergleich zur RDF-Quelldatei belegt die RCS-Datenbanksystem inklusive aller Indexe fast den doppelten Speicherplatz auf dem Sekundärspeicher. □

Zum einfacheren Erkennen der Zusammenhänge zwischen den Messwerten werden die Datensatzgrößen und der Speicherplatzverbrauch in Abbildung 6.4 graphisch dargestellt.

Beobachtung 6.2. Die Daten zeigen, dass abgesehen vom 10k-Datensatzes der Speicherplatzbedarf proportional zur Datensatzgröße steigt. Beispielsweise beträgt das Verhältnis der Tripelanzahlen zwischen dem 100k- und 500k-Datensatzes 5,2 und das Verhältnis der belegten Datenseiten 4,1. Bei den beiden nächstgrößeren Datensätzen sind die jeweiligen Verhältnisse 2,1 und 1,8. Aus Tabelle 6.7 erkennt man darüber hinaus, dass die durchschnittliche Anzahl der pro Datenseite gespeicherten Tripel mit wachsender Größe des Datensatzes steigt. Schließlich zeigt sich im Vergleich mit Jena TDB, dass der RCS für den jeweiligen Datensatz ungefähr doppelt so viel Speicherplatz benötigt. □

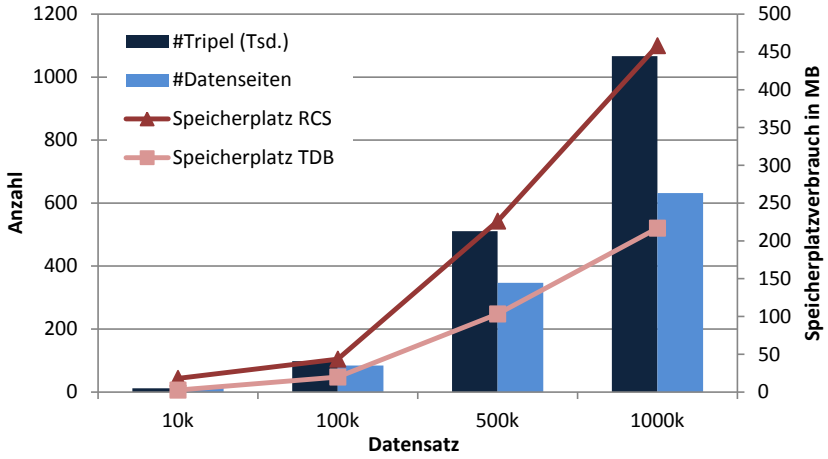


Abbildung 6.4: Datensatzgrößen und Speicherplatzverbrauch

Datenverteilung. Beim Speichern von RDF-Tripeln im RCS werden zum einen Tripel auf Datenseiten abgelegt und zum anderen auch Einträge in den Subjekt-, Prädikat- und Objektindexen erstellt. Im Folgenden werden Histogramme über die Verteilung der Daten nacheinander präsentiert.

In Abbildung 6.5 werden zunächst die Verteilung der Subjekte auf die Datenseiten für alle Datensätze dargestellt (in Prozent aller Datenseiten, vgl. Tabelle 6.7).

Beobachtung 6.3. Der überwiegende Teil der Datenseiten beinhalten bis zu 20 oder zwischen 180 und 200 Subjekte, wobei die zweite Gruppe in Summe einen höheren Anteil einnimmt. □

Abbildung 6.6 zeigt die Verteilung der Tripel auf die Datenseiten für alle Datensätze, die wiederum als Prozentzahl aller Datenseiten angegeben wurden.

Beobachtung 6.4. Bis auf den Datensatz 10k ist eine deutliche Tendenz zu Datenseiten mit ca. 1900 Tripeln zu erkennen. □

Mit den beiden Diagrammen in Abbildung 6.7 werden die Prädikat- und Objektindexe näher betrachtet. Bei diesem Index ist die Anzahl der gesetzten

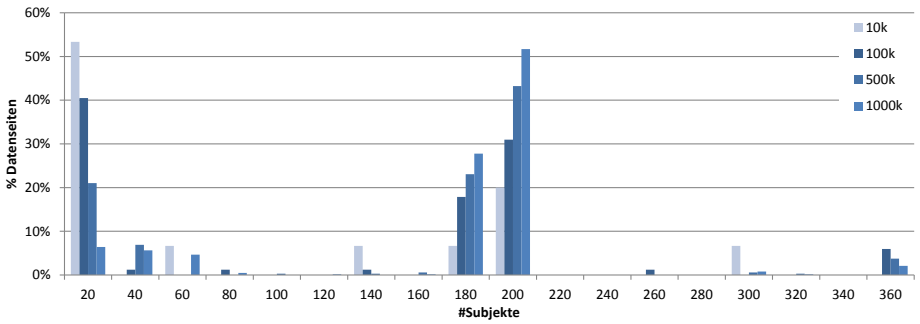


Abbildung 6.5: Prozentzahl aller Datenseiten mit n Subjekten

Datensatz	#Bitsets		Max. gesetzte Bits		Bitsetgröße
	I_p	I_o	I_p	I_o	(#Subjekte)
10k	40	3.672	1.244	601	1.244
100k	40	24.734	9.875	5.000	9.875
500k	40	114.210	49.052	28.000	49.052
1000k	40	224.242	99.517	60.000	99.517

Tabelle 6.8: Gesamtzahl an Bitsets, maximale Anzahl gesetzter Bits und Bitsetgröße für Prädikat- und Objektindexe

Bits in den Bitsets von besonderem Interesse, da diese den Speicherplatzbedarf der Bitsets bestimmen (vgl. Abschnitt 3.3.2). Die Bitsetgröße entspricht der Anzahl an Subjekten und kann Tabelle 6.8 entnommen werden. Die Abbildungen stellen für die beiden Indexe den prozentualen Anteil an allen Bitsets dar, bei denen n Prozent des Bitsets gesetzt sind. Betrachtet man beispielsweise Abbildung 6.7a den Balken bei 50 % gesetzer Bits, dann hat der dortige Balken die folgende Bedeutung: 17,5 % aller Bitsets in der Datenbank des 100k-Datensatzes (7 von 40 Bitsets) haben ca. 50 % aller Bits gesetzt sind (5.000 von 9.875 Bits).

Beobachtung 6.5. Wie aus Tabelle 6.8 ersichtlich ist, ist die Anzahl der distinkten Prädikate für alle Datensätze gleich. In allen Datensätzen werden die meisten Prädikate (19 von 40) eher selten verwendet – nur wenige Bits sind in den entsprechenden Bitsets gesetzt (vgl. Abbildung 6.7a). Dahingegen gibt es nur wenige Prädikate (3 von 40), bei denen alle Bits im Bitset gesetzt sind. Insgesamt gilt für die verwendeten Datensätze, dass die in wenigstens 75 % aller Bitsets höchstens 30 % der maximalen Anzahl an gesetzten Bits gesetzt sind. Dies bedeutet, dass ein großer Teil der Bitsets eher schwach besetzt ist. □

Ein anderes Bild zeigt der Objektindex in Abbildung 6.7b, das mit der folgenden Beobachtung zusammengefasst wird:

Beobachtung 6.6. Fast alle Bitsets haben nur sehr wenige gesetzte Bits.[2] Tabelle 6.8 zeigt für den Objektindex, dass die Anzahl der distinkten Werte mit der

[2]Das Diagramm zeigt nur einen Ausschnitt, da im nicht gezeigten Teil zu wenige Bitsets mit der entsprechenden Anzahl an gesetzten Bits existieren, um einen sichtbaren Balken zu erzeugen.

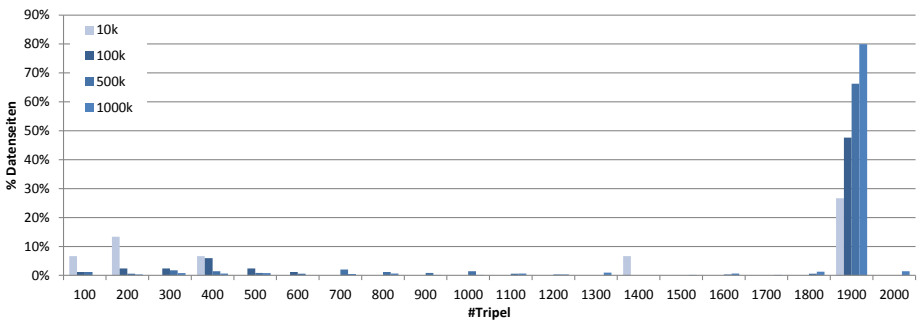
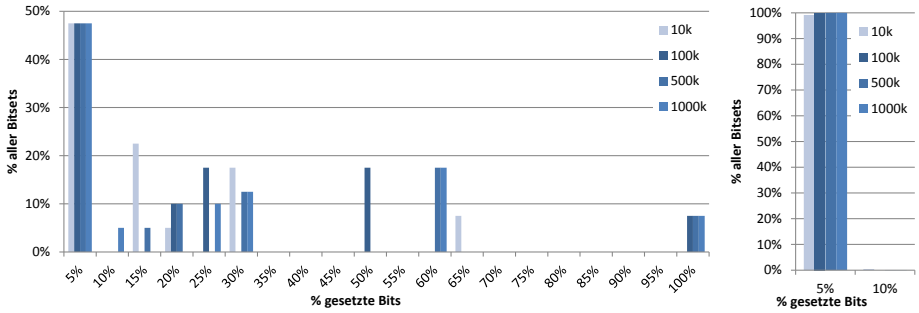


Abbildung 6.6: Prozentzahl aller Datenseiten mit n Tripeln

Größe des Datensatzes wächst. Im Gegensatz zum Prädikatindex gibt es kein Objekt, das in Kombination mit allen Subjekten vorkommt. Wie der Tabelle auch entnommen werden kann, beträgt die maximale Anzahl an gesetzten Bits zwischen 50 und 60 % der Bitsetgröße. □



(a) Prädikatindex (b) Objektindex

Abbildung 6.7: Histogramm der Prädikat- und Objekindexe über die Prozentzahl aller Bitsets mit n % der maximal gesetzten Bits

Obwohl für jedes Prädikat und jedes Objekt ein Bitset angelegt wird (vgl. Tabelle 6.8), ist der benötigte Speicherplatz sehr gering. Tabelle 6.9 listet die Gesamtgröße (in KByte) des durch die Bitsets von Prädikat- und Objektindex belegten Speicherplatzes auf. Diese zeigt auch die maximale Größe (in Bytes) eines Bitsets. Weiteren Speicherplatz verbraucht der Normalisierungsindex zum Konvertieren von Literalen und der IRIs von Ressourcen zu Identifikatoren. Der dafür belegte Speicherplatz wird in Tabelle 6.9 dargestellt.

Beobachtung 6.7. Die Bitsets der Prädikat- und Objektindexe nehmen im Vergleich zu den Datenseiten sehr wenig Speicherplatz in Anspruch (vgl. Tabelle 6.7), während der Normalisierungsindex ca. 13 % des gesamten Speicherplatzverbrauchs ausmacht. □

Datensatz	$\sum \|\|I_p\|\|$ in KB	$\max \|\|I_p\|\|$ in B	$\sum \|\|I_o\|\|$ in KB	$\max \|\|I_o\|\|$ in B	Norm.-Index #Einträge	Norm.-Index Werte (MB)
10k	0,5	45	15,1	42	4.655	0,7
100k	3,4	275	123,7	323	32.633	5,4
500k	11,1	1.234	632,9	1785	156.597	27,6
1000k	19,4	2.387	1.297,6	3816	314.244	56,9

Tabelle 6.9: Gesamtgröße aller Bitsets und maximale Größe eines Bitsets von Prädikat- und Objektindex sowie Anzahl der Einträge und Größe der Werte im Normalisierungsindex

Ladezeit. Obwohl für die Evaluation des RCS die Ausführungszeiten von Anfragen von größerem Interesse sind, wurden auch die Ladezeiten für die einzelnen Datensätze gemessen. Abbildung 6.8 gibt die Ladezeiten und die Anzahl geladener Tripel pro Sekunde für den RCS und Jena TDB wieder.

Beobachtung 6.8. Die Messwerte zeigen, dass mit wachsendem Datensatz der Durchsatz im RCS sinkt und im Vergleich mit Jena TDB deutlich langsamer ist (Faktor 4,5 beim 10k- und 17,0 beim 1000k-Datensatz). Besser als in der tabellarischen Darstellung kann man in dem Diagramm in Abbildung 6.8 erkennen, dass die Ladezeiten im RCS nicht proportional zur Tripelanzahl steigt – sie haben eher einen leicht exponentiellen Character. □

Datensatz	RCS		TDB	
	s	$\lvert\mathcal{T}\rvert/_s$	s	$\lvert\mathcal{T}\rvert/_s$
10k	10,8	1.105	2,4	4.957
100k	84,3	1.160	10,4	9.436
500k	536,6	952	41,3	12.363
1000k	1.355,1	787	79,8	13.375

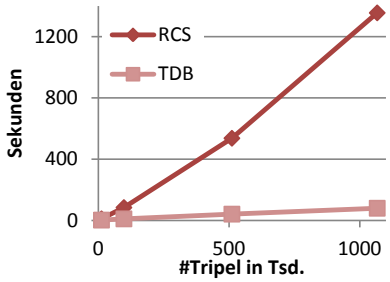


Abbildung 6.8: Ladezeiten aller Datensätze für RCS und TDB

6.3.2.2 Sternförmige Anfragen

Im Folgenden werden die Ausführungszeiten von Anfragegruppe aufgelistet, in denen nach Möglichkeit nur eine von den am Anfang dieses Abschnitts eingeführten Eigenschaften I–IV variiert. Dadurch kann der Einfluss der jeweiligen Eigenschaft auf die Ausführungszeit genauer bestimmt werden. Zunächst werden die Ergebnisse der Anfragen mit konstantem Subjekt vorgestellt. Anschließend werden Anfragen mit einer Variablen als Subjekt betrachtet. Die Anfragen und die Messwerte können dem Anhang A.1 entnommen werden.

Konstantes Subjekt. Abbildung 6.9 stellt die Ausführungszeiten einer Anfragegruppe dar, in der die Anzahl der Tripelmuster (1, 2, 4 und 6) verändert wurde (Eigenschaft II). Alle Anfragen hingegen haben eine Konstante als Subjekt, beinhalten genau eine Variablen und liefern ein Ergebnis zurück. Hieraus folgt zwangsläufig, dass die Anzahl der Konstanten mit wachsender Anzahl an Tripelmustern steigt. Zum Vergleich ist die Ausführungszeit der Anfragen für Jena TDB als Linie im Graph eingezeichnet – ein Datenpunkt entspricht der durchschnittlichen Ausführungszeit über alle Datensatzgrößen hinweg.

Beobachtung 6.9. Der Graph zeigt, dass sich die Ausführungszeiten der Anfragen für die verschiedenen Anzahlen an Tripelmustern ungefähr gleich verhalten. Nur für die Datensätze 500k und 1000k sind die Ausführungszeiten etwas höher. □

Der nächste Graph in Abbildung 6.10 zeigt die Ausführungszeiten einer Anfragegruppe mit denselben Eigenschaften wie zuvor. Der Unterschied besteht vor allem in der Größe des Anfrageergebnisses – alle Anfragen liefern 11 Ergebnisse zurück. Dieses wurde durch das Einführen einer weiteren Variablen erreicht. Der Graph zeigt keine Ergebnisse für sechs Tripelmuster, weil aufgrund der Eigenschaften des Datensatzes keine Anfrage mit elf Ergebnissen konstruiert werden konnte.

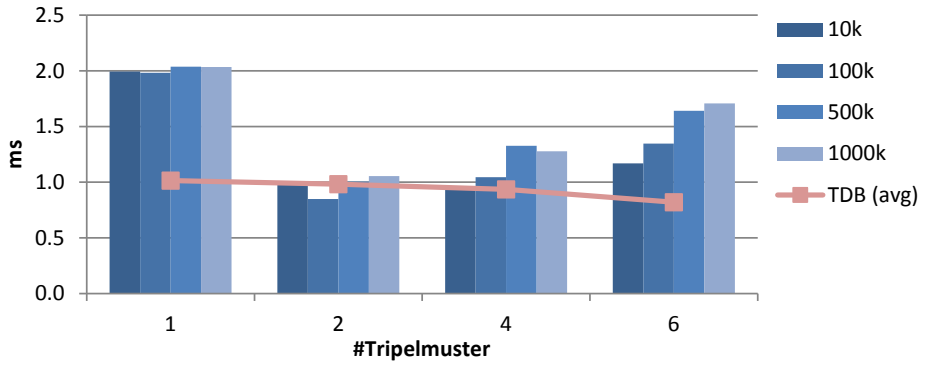


Abbildung 6.9: Ausführungszeiten für Anfragen mit konstantem Subjekt, wachsender Tripelmusteranzahl und einem Ergebnis

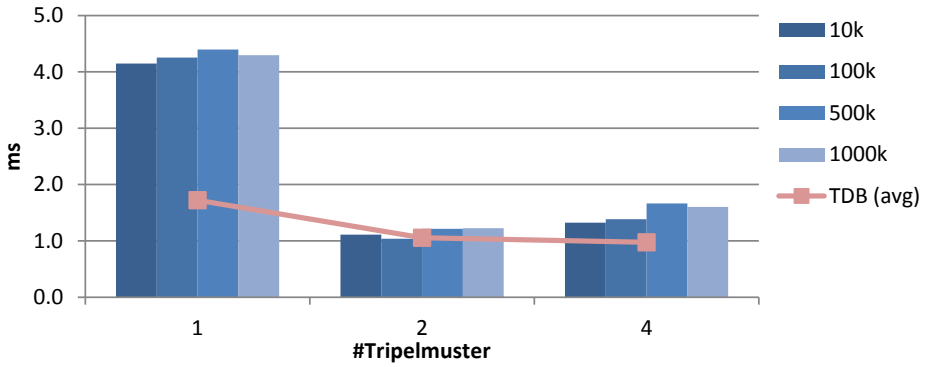
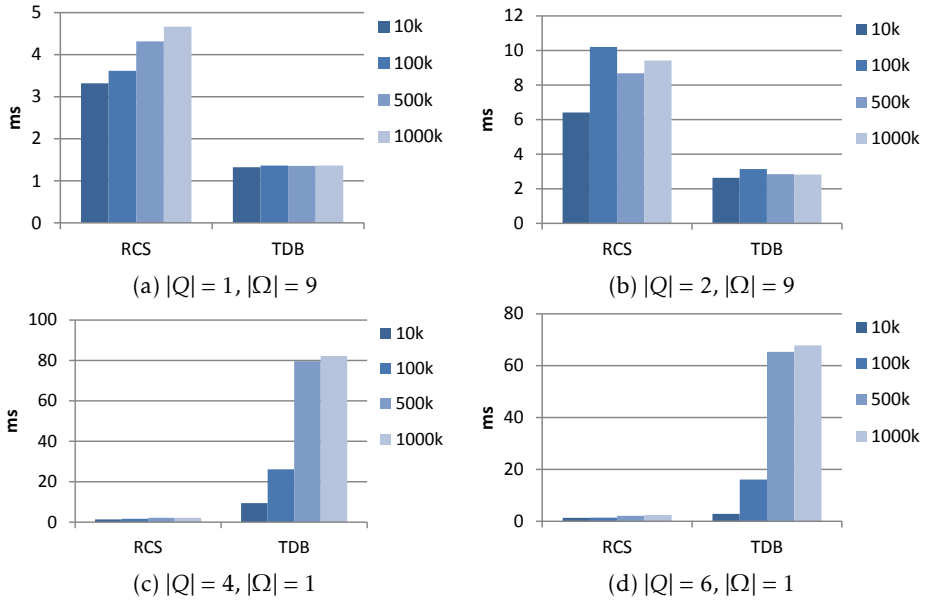


Abbildung 6.10: Ausführungszeiten für Anfragen mit konstantem Subjekt, wachsender Tripelzahl und elf Ergebnissen

Beobachtung 6.10. Im Vergleich zu der vorherigen Anfragegruppe (vgl. Abbildung 6.9) haben sich die Ausführungszeiten nur für die Anfragen mit einem Tripelmuster wesentlich verschlechtert (verdoppelt), die Anfragen mit mehr Tripelmustern ergeben sehr ähnliche Ausführungszeiten wie zuvor. Die Ausführungszeiten für Jena TDB zeigen ein ähnliches Verhalten. □

Beobachtung 6.11. Für alle in den Abbildungen 6.9 und 6.10 aufgeführten Anfragen wurde ein Mal auf den Subjektindex zugegriffen und eine Datenbankseite gelesen. □

Variable als Subjekt. Abbildung 6.11 zeigt den ersten Teil der Experimente mit sternförmigen Anfragen, die eine Variable als Subjekt haben (Eigenschaft I). Die Diagramme zeigen jeweils das Ausführen einer Anfrage mit einer bestimmten Anzahl an Tripelmustern (1, 2, 4 und 6) und einem kleinen Ergebnis (Eigenschaften II und IV). Beispielsweise zeigt das Diagramm 6.11a die Ausführungszeiten für eine Anfrage mit einem Tripelmuster und neun Ergebnissen. Zum Vergleich wurden jeweils auch die Ausführungszeiten für Jena TDB dargestellt.



(a) $|Q| = 1, |\Omega| = 9$ (b) $|Q| = 2, |\Omega| = 9$

(c) $|Q| = 4, |\Omega| = 1$ (d) $|Q| = 6, |\Omega| = 1$

Abbildung 6.11: Ausführungszeiten für Anfragen mit einer Variablen als Subjekt und wenigen Ergebnissen

Beobachtung 6.12. Die Diagramme zeigen die Tendenz, dass für größere Datensätze mehr Zeit zum Ausführen von Anfrage mit wenigen Tripelmustern benötigt wird. Im Gegensatz dazu beträgt die Ausführungszeit für die größeren Anfragen nur wenige Millisekunden und der Unterschied zwischen den verschieden Datensätzen ist gering. □

Obwohl Jena TDB in Abbildung 6.11 nur als Referenzsystem dargestellt wird, ist im Zusammenhang der Evaluation des RCS-Speichermodells folgende Beobachtung von besonderem Interesse:

Beobachtung 6.13. Während in Jena TDB die Ausführungszeiten bei den Anfragen mit 4 und 6 mit größer werdendem Datensatz steigen, bleiben diese im RCS nahezu unverändert. □

Für die in Abbildung 6.11 verwendeten Anfragen stellt die nebenstehende Tabelle 6.10 die Zugriffsstatistiken dar. Die erste Spalte beinhaltet die Anzahl der Tripelmuster der Anfragen, die folgenden drei Spalten beinhalten die Anzahlen der Zugriffe auf die Subjekt-, Prädikat- und Objektindexe auf. Die letzten beiden Spalten zeigen schließlich, wie viele Datenseiten insgesamt gelesen wurden und wie viele davon zur Lösung der Anfrage beitrugen.

$\|Q\|$	#Zugriffe				
	I_s	I_p	I_o	$\|p_r\|$	$\|p_\Omega\|$
1	0	1	1	1	1
2	0	2	1	1	1
4	0	4	3	1	1
6	0	6	5	1	1

Tabelle 6.10: Zugriffe auf Indexe und Datenseiten für alle Datensätze (zu Abb. 6.11)

Beobachtung 6.14. Für alle Anfragen wurden ausschließlich die Prädikat- und Objektindexe verwendet, wobei die Anzahl der Zugriffen der Anzahl der Konstanten

in der jeweiligen Position entsprechen. Aufgrund der Verwendung der Indexe wurde genau eine Seite selektiert und gelesen – siehe Tabelle 6.4 für die Gesamtzahl der Datenseiten eines Datensatzes. □

Die nächste Anfragegruppe wurde ähnlich zu der vorherigen konstruiert, jedoch liefern die Anfragen ein größeres Ergebnis zurück, d.h. Eigenschaft IV wurde geändert. Die Ergebnisgrößen für die jeweiligen Anfrage- und Datensatzgrößen sind in Tabelle 6.11 dargestellt. Da die größeren Anfragemuster selektiver sind, verringert sich die Ergebnisgröße.

Datensatz	|Q|			
	1	2	4	6
10k	1.244	1.274	202	213
100k	9.875	10.125	1.695	1.764
500k	49.052	50.452	9.787	7.462
1000k	99.517	102.517	21.027	7.462

Tabelle 6.11: Ergebnisgrößen der Anfragen (zu Abb. 6.12)

Die Ausführungszeiten zum Berechnen der Lösungen für diese Anfragegruppe werden in den Diagrammen von Abbildung 6.12 dargestellt.

Beobachtung 6.15. Die Diagramme zeigen grundsätzlich höhere Ausführungszeiten für größer werdende Datensätze. Für die Anfragen mit sechs Tripelmustern kann man feststellen, dass die Ausführungszeiten für die Datensätze 500k und 1000k ungefähr gleich hoch sind (vgl. Abbildung 6.12d). □

Um das Verhalten der ähnlichen Ausführungszeiten eingehender untersuchen zu können, zeigt Abbildung 6.13 die pro Millisekunde generierte Anzahl von Ergebnissen (Durchsatz) für RCS und Jena TDB. Für Anfragen mit einem Tripelmuster beträgt der Durchsatz ca. 50 Ergebnisse/ms während er für die Anfragen mit sechs Tripelmustern nur 15 Ergebnisse/ms erreicht.

Beobachtung 6.16. Lässt man den kleinsten Datensatz außer acht, dann ist der erzielte Durchsatz nicht von der Größe des Datensatzes abhängig sondern vielmehr von der Größe des Tripelmusters. □

Wie das Diagramm in Abbildung 6.14 illustriert, verhält sich das Vergleichssystem Jena TDB ähnlich zum RCS. Vergleicht man jedoch das Verhältnis zwischen den Durchsätzen von TDB und RCS, so kann folgende Beobachtung dem Diagramm entnommen werden:

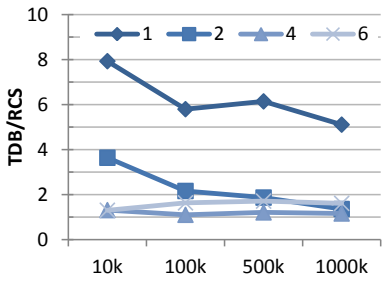
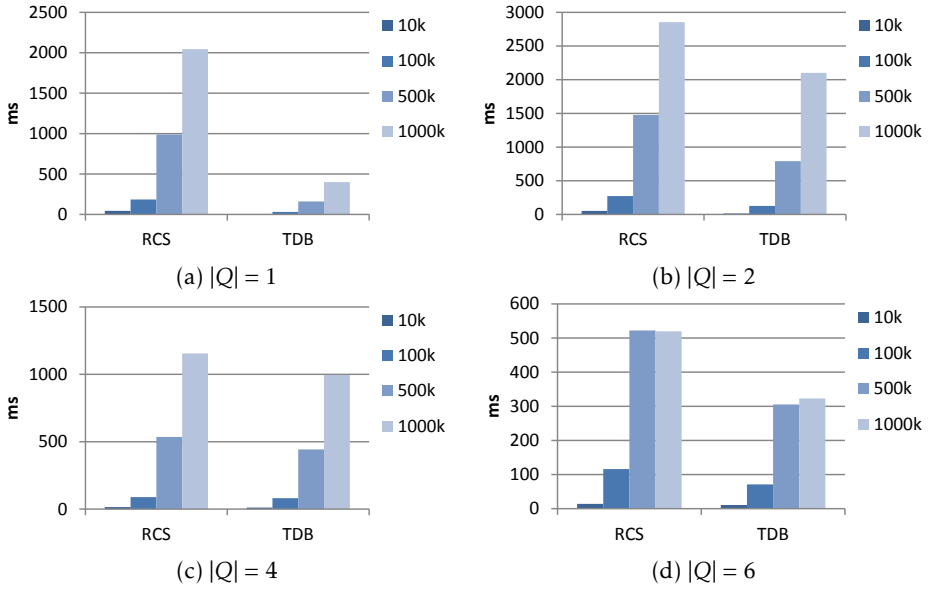


Abbildung 6.14: Verhältnis der Durchsätze von RCS und TDB

Beobachtung 6.17. Während Jena TDB für den kleinsten Datensatz einen vielfach höheren Durchsatz erreicht (durchschnittlicher Faktor für 10k ist 6,2), sinkt dieser bei den größeren Datensätzen nahezu auf das Niveau des RCS ab (durchschnittliche Faktoren für 500k und 1000k sind 1,2 bzw. 1,6). □



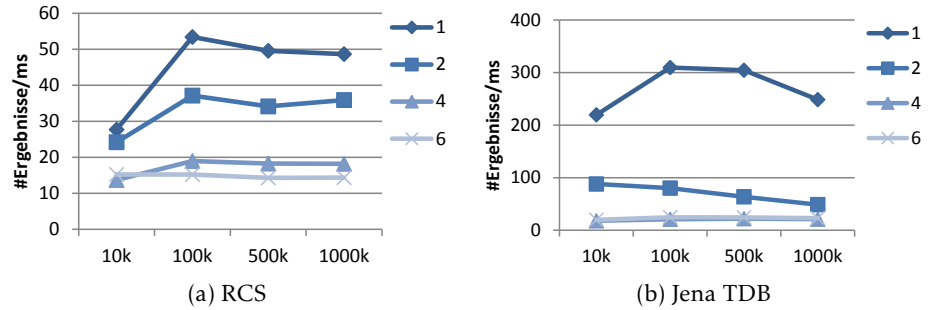
(a) $|Q| = 1$ (b) $|Q| = 2$

(c) $|Q| = 4$ (d) $|Q| = 6$

Abbildung 6.12: Ausführungszeiten für Anfragen mit einer Variablen als Subjekt und vielen Ergebnissen

Schließlich zeigt Tabelle 6.12 die Zugriffsstatistiken für die Anfragegruppe mit einer Variablen als Subjekt und großem Ergebnis. Analog zu den Anfragen mit wenigen Ergebnisses werden wiederum nur die Prädikat- und Objektindexe und nicht der Subjektindex verwendet. Darüber hinaus kann Folgendes beobachtet werden:

Beobachtung 6.18. Im Vergleich zur vorherigen Anfragegruppe (vgl. Tabelle 6.10) werden für den überwiegenden Teil der Anfragen mehr Datenseiten zugegriffen. □



(a) RCS (b) Jena TDB

Abbildung 6.13: Anzahl der generierten Ergebnisse pro Millisekunde für Anfragen unterschiedlicher Größe

$\|Q\|$	#Zugriffe			10k		100k		500k		1000k	
	I_s	I_p	I_o	$\|P_r\|$	$\|P_\Omega\|$	$\|P_r\|$	$\|P_\Omega\|$	$\|P_r\|$	$\|P_\Omega\|$	$\|P_r\|$	$\|P_\Omega\|$
1	0	1	0	15	15	84	84	347	347	623	623
2	0	2	0	15	15	84	84	347	347	623	623
4	0	4	0	2	2	2	2	81	81	174	174
6	0	6	1	2	2	2	2	62	62	62	62

Tabelle 6.12: Zugriffe auf Indexe und Datenseiten aufgegliedert nach Datensätzen (zu Abb. 6.12)

6.3.2.3 Anfragen mit Join-Operation

Im Folgenden werden die Ergebnisse der Anfragen vorgestellt, die eine Join-Operation zwischen zwei sternförmigen Graphmustern beinhalten (vgl. Abschnitt 6.3.1). Der Fokus der Experimente in diesem Abschnitt liegt auf dem Untersuchen des grundsätzlichen Verhaltens des RCS, wenn eine Join-Operation im Anfrageausführungsplan enthalten ist. Da aufgrund des auf Datenseiten basierenden Speichermodells die im Kontext von RDBMS etablierten Join-Algorithmen verwendet werden (Abschnitt 5.4.5), wird in dieser Arbeit eine Evaluation von verschiedenen Join-Algorithmen verzichtet.

Die Abbildungen 6.15 bis 6.17 geben einen Überblick über die Ausführungszeiten für die drei Möglichkeiten des Joins von Graphmustern: gemeinsame Objekt-Variable, gemeinsame Objekt-Konstante und Subjekt des einen Graphmusters ist ein Objekt des anderen. Jede Gruppe Anfragen wurde für alle Datensätze ausgeführt (s. Abschnitt *Evaluationsumgebung* auf Seite 149). Um für die mit RCS gemessenen Ausführungszeiten einen Bezugspunkt zu haben, wurden in den Diagrammen auch die jeweiligen Ausführungszeiten mit Jena2 TDB mit dargestellt.

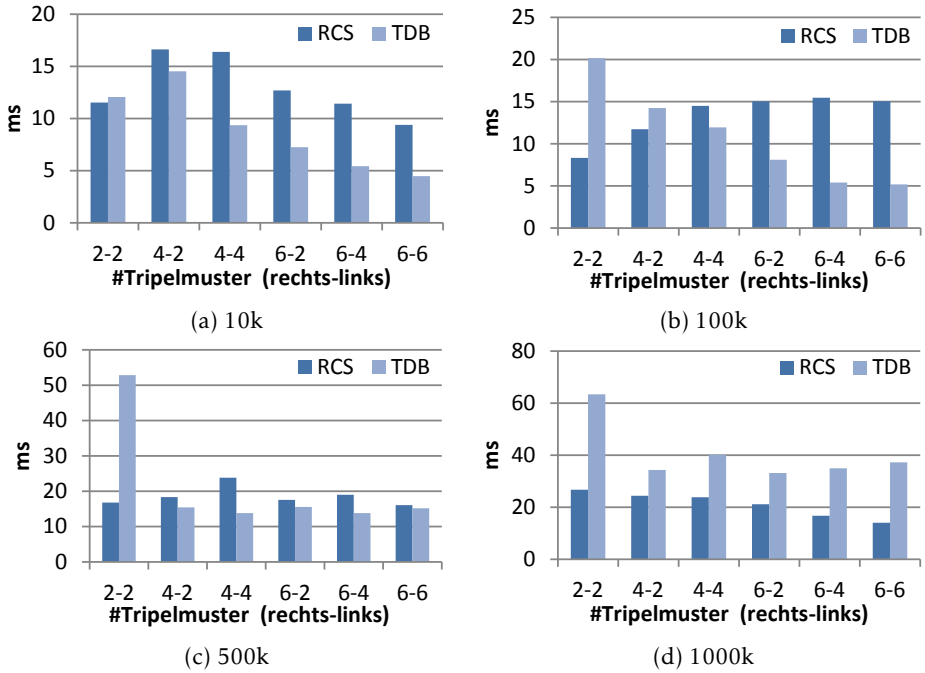
Beobachtung 6.19. Im Falle einer gemeinsamen Variablen als Objekt zeigt Abbildung 6.15, dass sich für die verschiedenen Datensätze unterschiedliche Kurven ergeben. Beim 10k-Datensatz haben die Anfragen 4–2 und 4–4 die höchste Ausführungszeit, bei 100k die Anfragen 4–4 bis 6–6, bei 500k die Anfrage 4–4 und bei 100k die Anfrage 2–2. Bis auf den 100k-Datensatz zeigt sich die Tendenz, dass die Anfragen mit großen Teilmustern in kürzerer Zeit ausgeführt werden. □

Die Diagramme in Abbildung 6.16 präsentieren die Ausführungszeiten, wenn der Join über ein konstantes Objekt berechnet wird.

Beobachtung 6.20. Über alle Datensätze hinweg kann eine annähernd gleiche Ausführungszeit festgestellt werden. Ausgenommen des 100k-Datensatzes kann aus den Diagrammen die Tendenz abgelesen werden, dass die Anfragen mit größeren Teilmustern eine höhere Ausführungszeit haben. □

In Abbildung 6.17 werden die Ausführungszeiten für einen Join dargestellt, bei dem das Subjekt des einen Graphmusters ein Objekt des anderen ist. Zunächst kann folgende Beobachtung festgehalten werden:

Beobachtung 6.21. Unabhängig von den Größen der Teilmuster korreliert die Ausführungszeit mit der Größe des Datensatzes: Je größer der Datensatz ist desto höher ist die Ausführungszeit. □



(a) 10k

(b) 100k

(c) 500k

(d) 1000k

Abbildung 6.15: Ausführungszeiten der Join-Variante OV für RCS und TDB

Über die verschieden Datensätze hinweg kann aus den Experimenten keine klare Aussage über die Zusammenhänge zwischen Ausführungszeit und Anfrage getroffen werden. Beispielsweise ist die Ausführungszeit für die Anfrage 4–2 bei den beiden kleineren Datensätzen besonders hoch, während sie bei den anderen beiden Datensätzen eher gering ist.

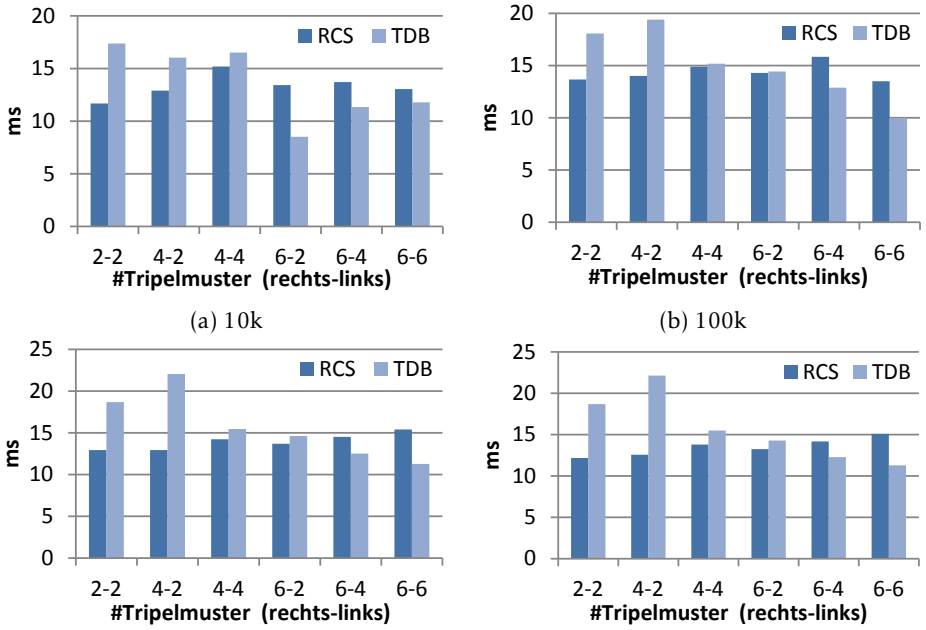
Beobachtung 6.22. Betrachtet man die Ausführungszeiten des Referenzsystems Jena TDB, dann lässt sich auch bei diesem kein konsistentes Verhalten der Ausführungszeiten feststellen. □

Beispielsweise ist in Abbildung 6.15b über die Anfrage 2–2 bis hin zu 6–6 eine ständige Abnahme der Ausführungszeit beobachten, während in Abbildung 6.15c die Ausführungszeit für 2–2 besonders hoch und für die übrigen annähernd gleich ist.

Tabelle 6.13 zeigt die Anzahl der Lesezugriffe auf Datenseiten im RCS, die zum Berechnen der Gesamtlösung erforderlich waren. Dabei wird nicht unterschieden, ob dieselbe Seite mehrfach zugegriffen worden ist. Bezüglich der Anzahl der Indexzugriffe wurden dasselbe Verhalten wie bei der Ausführung der sternförmigen Anfragen beobachtet. Für jedes konstante Prädikat und Objekt wurde der Prädikat- bzw. Objektindex genau einmal zugegriffen. Der Subjektindex wurde für keine Anfrage verwendet.

Join-Art	10k	100k	500k	1000k
OV	4	10	4	2
OK	2	2	3	5
SV	6	38	110	160

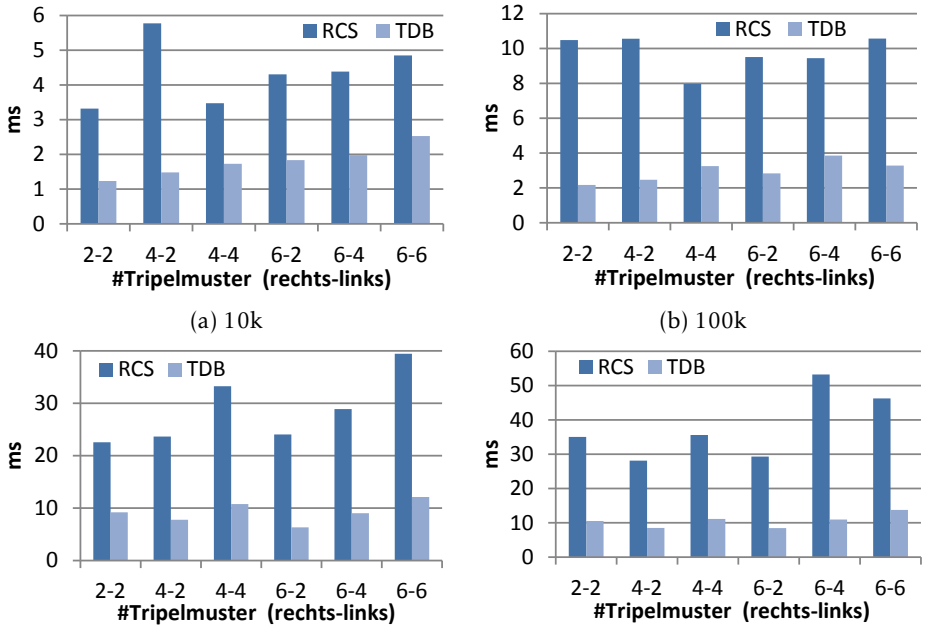
Tabelle 6.13: Anzahl der Datenseitenzugriffe für alle Datensätze



(a) 10k (b) 100k

(c) 500k (d) 1000k

Abbildung 6.16: Ausführungszeiten der Join-Variante OK für RCS und TDB



(a) 10k (b) 100k

(c) 500k (d) 1000k

Abbildung 6.17: Ausführungszeiten der Join-Variante SV für RCS und TDB

Beim Vergleich der Ausführungszeiten zwischen RCS und Jena TDB ist hingegen Folgendes besonders auffällig:

Beobachtung 6.23. Während die Ausführungszeiten der beiden Systeme bei den Join-Arten OV und OK miteinander vergleichbar sind, so sind sie bei Jena TDB im Falle eines SV-Joins deutlich geringer als beim RCS. □

6.3.3 Diskussion und Schlussfolgerungen

In diesem Abschnitt werden die Ergebnisse der Experimente zusammengeführt und Schlussfolgerungen für das RCS-Speichermodell gezogen. Die Diskussion gliedert sich dabei in die beiden Teile Datenhaltung und Anfrageausführung. Der erste Teil beschäftigt sich mit der physischen Speicherung der RDF-Daten, während sich der zweite eher mit den Auswirkungen des Speicherkonzepts – Gruppierung anhand des Subjekts – auf die Anfrageausführung befasst.

6.3.3.1 Datenhaltung

Während der Evaluation wurden Messwerte zum Speicherplatzverbrauch, den Ladezeiten und der Belegung der Indexe gesammelt, anhand derer die Speicherplatzeffizienz des RCS-Speichermodells im Folgenden beurteilt wird.

In Beobachtung 6.1 wurde eine knappe Verdoppelung des Speicherplatzbedarfs im Vergleich zur RDF-Quelldatei und zu Jena TDB festgestellt. Dieser hohe Speicherplatz wird im Wesentlichen durch die Berkeley-DB verursacht, die zur Verwaltung der Datenseiten und Zugriffsstrukturen verwendet wird. Auf Grundlage der in den Experimenten gemessenen Werte gibt Tabelle 6.14 einen Überblick über den minimal benötigten Speicherplatz für die jeweiligen Komponenten im RCS wider. In der letzten Zeile der Tabelle zeigt den prozentualen Anteil bezüglich des tatsächlich vom RCS-Datenbanksystem belegten Speicherplatzes. Da die Tabelle nur den von den eigentlichen Werten belegten Speicherplatz darstellt, wird der restliche Anteil für Zugriffsstrukturen der Berkley-DB belegt – dies sind ca. 84 % bei den drei größten Datensätzen. Diese Berechnungen lassen den Schluss zu, dass mit einer speziell auf die Bedürfnisse des RCS abgestimmte Komponente zur Verwaltung von Datenseiten und Zugriffsstrukturen ein deutlich geringerer Speicherplatzbedarf erreicht werden kann.

Unabhängig von dem tatsächlich belegten Speicherplatz lässt Beobachtung 6.2 den Schluss zu, dass Speicherplatzbedarf proportional zu der Anzahl der Tripel im RDF-Graphen wächst. Da für jedes distinkte Prädikat und Objekt ein separater Bitset angelegt wird, hängt der Speicherplatzbedarf auch von deren Anzahl ab. Aufgrund des Verwendens eines komprimierten Bitsets belegen diese jedoch so wenig Speicherplatz (vgl. Beobachtungen 6.6), dass dieser nicht ins Gewicht fällt. Das Verhalten des RCS korrespondiert somit zu dem Vergleichssystem Jena TDB.

In Beobachtung 6.4 wurde für die drei größten Datensätze eine Belegung von rund 1900 Tripeln pro Datenseite festgestellt. In der theoretischen Analyse des Speichermodells wurde eine Formel für die maximale Anzahl der pro Datenseite speicherbaren Tripel und Beispiele für verschiedene Konfigurationen angegeben. In der für die Experimenten verwendeten Konfiguration (16 KB

Komponente	10k	100k	500k	1000k
Datenseiten	240,0	1.344,0	5.552,0	9.968,0
Subjektindex	9,7	77,1	383,2	77,5
Prädikatindex	0,5	3,4	11,1	19,4
Objektindex	15,1	123,7	632,9	1.297,6
Norm.-Index	718,4	5.676,6	28.895,7	59.530,1
Summe	983,7	7.225,0	35.474,9	71.592,6
Summe (MB)	1,0	7,1	34,6	69,9
Anteil (%)	5,4	16,6	15,7	15,6

Tabelle 6.14: Minimaler Speicherplatzverbrauch (in KB) für Datenseiten und Zugriffsstrukturen sowie den Anteil am tatsächlich belegten Speicherplatz

Datenseite und 4 Byte pro ID) wurde ein Maximum von ungefähr 2000 Tripel berechnet (vgl. Tabelle 3.2 auf Seite 39). Obwohl in dem Speichermodell die Tripel zusätzlich nach dem Wert des Prädikats `rdf:type` auf Datenseiten gruppiert wurden, konnte eine gutes Verhältnis zwischen Tripelanzahl und Datenseitengröße erreicht werden. Nur bei kleinen Datensätzen ist das Verhältnis ungünstig und resultierte in einer verhältnismäßig großen Datenbank. Aus diesem Grund scheint eine Gruppierung anhand von `rdf:type` nicht sinnvoll. Insgesamt müssten Gruppierungskriterien von Subjekten in weiteren Experimenten eingehender untersucht werden.

Wie in Abschnitt *Zugriffsstrukturen* auf Seite 32 wurden für das effiziente Lokalisieren von Prädikaten und Objekten Bitsets als Zugriffsstrukturen eingeführt. In der theoretischen Analyse wurde die Behauptung aufgestellt, dass diese sehr wenig Speicherplatz einnehmen werden und im Hauptspeicher gehalten werden können. Beobachtung 6.5 und 6.6 über die zumeist schwach besetzten Bitsets untermauern diese Behauptung. Damit ergibt sich die Schlussfolgerung, dass Bitsets ein speichereffizientes Mittel zum Lokalisieren von Prädikat- und Objektwerten dargestellt. In zukünftigen Untersuchungen zum RCS-Speichermodell sollte eine extensivere Nutzung von Bitsets in Betracht gezogen werden. Beispielsweise kann im Moment nur das Subjekt zu einem Prädikat mit Hilfe der Bitsets bestimmt werden, jedoch erscheint anhand der Beobachtungen auch eine Indexierung auf Tripelebene speichereffizient realisierbar.

Einen wesentlichen Anteil an dem Speicherplatzverbrauch (mind. 13 %) nimmt der Normalisierungsindex ein, der IRIs und Literale auf intern verwendete IDs abbildet (vgl. Beobachtung 6.7). Bei der Wahl von Verfahren zur Reduktion des Speicherplatzes dieses Indexes muss gleichzeitig dessen Verwendbarkeit für die Anfrageverarbeitung berücksichtigt werden. Bei der Entwicklung des RCS-Speichermodells in dieser Arbeit lag der Fokus auf dem grundsätzlichen Anfrageverhalten und nicht auf dem Auswerten von Filterausdrücken, die Literale beinhalten.

6.3.3.2 Ausführung von sternförmigen Anfragen

Eine wesentliche Hypothese bezüglich des RCS-Speichermodells ist, dass Anfragen mit konstantem Subjekt unabhängig von der Datensatzgröße mit kon-

stanter Komplexität beantwortet werden können. Beobachtungen 6.9 und 6.10 zeigen geringen Ausführungszeiten für diese Art von Anfragen und untermauern diese Behauptung. Auch die gemessene Anzahl für die Zugriffe auf Subjektindex und Datenseiten (vgl. Beobachtung 6.11) entsprechen den theoretischen Erwartungen.

Dahingegen sind die höheren Ausführungszeiten für die Anfragen mit einem Tripelmuster unerwartet und aus den folgenden Gründen nicht nachzuvollziehen: i) Unabhängig von der Anzahl der Tripelmuster und der Größe des Datensatzes wurde dieselbe Ressource als Subjekt verwendet, d.h. in allen Fällen wird dieselbe Seite zugegriffen. ii) Für alle Anfragen ist die Anzahl der Index- und Datenseitenzugriffe identisch. iii) Jede Anfrage wurde pro Datensatz zehn Mal ausgeführt, wobei die Ausführungszeiten in der jeweiligen Serie nur minimale Abweichungen aufweisen.

Die in Beobachtung 6.9 festgestellt geringfügig erhöhte Ausführungszeit für Anfragen mit einem Ergebnis bei den beiden größten Datensätzen konnte nicht bei den Anfragen mit elf Ergebnissen beobachtet werden (vgl. Abbildungen 6.9 und 6.10). Daher kann angenommen werden, dass diese durch andere Prozesse auf dem Evaluationsserver verursacht worden sind.

Die nächste Gruppe von untersuchten Anfragen beinhaltete eine Variable als Subjekt. Zunächst kann durch Beobachtung 6.14 die aufgrund der theoretische Analyse erhobene Behauptung bestätigt werden, dass zum einen die Anzahl der konstanten Werte ausschlaggebend für die Anzahl der Indexzugriffe ist und zum anderen der Subjektindex nicht verwendet wird.

Bezüglich der Ausführungszeiten wird als erste Beobachtung 6.12 diskutiert. Als Grund für die höhere Ausführungszeit der Anfragen mit 1 bzw. 4 Tripelmustern könnte man zunächst das größere Ergebnis (9 anstelle von 1) und einen daraus folgenden höheren Aufwand für das Generieren der Lösungsabbildungen annehmen. Dies kann jedoch nicht der Fall sein, da i) bei allen Anfragen dieser Serie genau eine Datenseite geladen wird (vgl. Tabelle 6.11) und ii) bei den Experimenten mit konstantem Subjekt keine derartig höhere Ausführungszeit zwischen Anfragen mit einem bzw. elf Ergebnissen festgestellt wurde.Aus diesen Gründen liegt die Vermutung nahe, dass die Jena-spezifische Anfragekomponente ARQ einen erheblichen Einfluss auf die Ausführungszeiten von Anfragen mit einer Variablen als Subjekt und mehreren Ergebnissen hat. In Folge dessen müsste eine speziell auf das RCS-Speichermodell abgestimmte Anfragekomponente entwickelt werden, um das tatsächliche Verhalten untersuchen zu können.

Beobachtung 6.13 verdeutlicht den erheblichen Nutzen der Prädikat- und Objektindexe für die Anfrageauswertung. Während in Jena TDB die Ausführungszeit einer Anfrage mit der Größe des Basis-Graphmusters korreliert, bleibt sie im RCS nahezu unverändert. Daraus folgt, dass die im RCS definierten Zugriffsstrukturen den Suchraum effektiv eingrenzen können.

Weitere Erkenntnisse zum Verhalten des RCS bei der Ausführung von sternförmigen Anfragen mit einer Variablen als Subjekt können aus den Beobachtungen 6.16 und 6.17. Obwohl die Ausführungszeit für Anfragen bei einem größeren Datensatz höher ist, verändert sich der Durchsatz (Anzahl der generierten Ergebnisse pro Millisekunde) unwesentlich. Daraus kann geschlussfolgert werden, dass im Wesentlichen nur die Effizienz der Implementierung der Datenverwaltung den erreichten Durchsatz bestimmt. Des Weiteren ergibt sich aus den Beobachtungen, dass der RCS Anfragen mit einem größeren

Basis-Graphmuster effizienter als Jena TDB verarbeiten kann – der erreichte Durchsatz sinkt langsamer. Wenn man außerdem die zuvor gewonnene Erkenntnis über den Einfluss von ARQ auf die Ausführungszeit hinzunimmt, wird der Einfluss des gewählten Software-Frameworks um so deutlicher und untermauert die Hypothese, dass eine spezialisierte Datenverwaltung und Anfrageverarbeitungskomponente zu Verbesserungen bei den Ausführungszeiten führen kann.

6.3.3.3 Ausführung von Join-Operationen

Anhand der Beobachtungen 6.19 bis 6.21 lässt sich feststellen, dass das Verhalten des RCS bei Join-Operationen von der Art der Verknüpfung der beiden sternförmigen Graphmuster abhängig ist. Join-Operationen über Objekte hinweg – Variable oder Konstante – zeigen ein recht einheitliches Bild: Tendenziell verringert sich die Ausführungszeit bei größere Graphmustern. Im Gegensatz dazu kann bei einem Subjekt-Objekt-Join keine klare Aussage getroffen werden.

Man betrachte die folgenden Fakten:

i) Die Größen der Zwischenergebnisse und des Endergebnisses des Joins waren für alle Anfrage innerhalb eines Datensatzes gleich (vgl. Tabelle 6.5 auf Seite 6.5).

ii) Bezüglich eines Datensatzes wurde dieselbe Anzahl von Datenseiten für alle Anfragen zugegriffen.

iii) Die größeren Graphmuster sind nur durch Hinzufügen eines Tripelmusters der Form `?s p ?o` entstanden.

Dann können die unterschiedlichen Ausführungszeiten nur durch die Implementierung des Join-Algorithmus oder externe Prozesse des Evaluationsrechners zurückgeführt werden. In dem Zusammenhang ist Beobachtung 6.22 interessant, in der aus den Experimenten auch für das Vergleichssystem kein einheitliches Verhalten abgeleitet werden kann. Die letzten beiden Beobachtungen 6.22 und 6.23 unterstützt die Vermutung, dass der verwendete Join-Algorithmus einen erheblichen Einfluss auf die Ausführungszeit hat und auf die jeweilige Join-Art spezialisierte Join-Algorithmen entwickelt und untersucht werden sollten.

6.4 Evaluation der Indexierung

In Kapitel 4 wurde das Indexieren eines RDF-Graphen auf Basis von Basis-Graphmustern (BGP) beschrieben. Dabei werden die Lösungen des Basis-Graphmusters (Indexmuster) persistiert und geeignet in den Anfrageausführungsplan integriert. Zur Evaluation dieses Ansatzes wurde im Laufe dieser Arbeit ein Indexmanager für BGP entwickelt, der die Lösungen eines Indexmusters in einer relationalen Datenbank abspeichert (vgl. Abbildung 6.2).

Das Ziel der nachfolgend beschriebenen Evaluation war herauszufinden, inwiefern sich ohne Veränderung des zugrunde liegenden Speichermodells die

Ausführungszeit unter Verwendung von graphmuster-basierten Indexen verändert. Dazu wurde die Anfragebearbeitung des RDF-Datenbanksystems Jena TDB [Jen10] so verändert, dass derartige Indexe für geeignete Teile einer Anfrage verwendet werden (vgl. *Überdeckung in Abschnitt 4.3.2*). Das heißt, der Optimierer ersetzt im Ausführungsplan einen BGP-Operator durch einen Indexoperator bzw. durch einen Join zwischen einem BGP-Operator und einem Index-Operator (vgl. Abschnitt *Heuristiken zum Generieren von alternativen Ausführungsplänen* auf Seite 117).

Im vorherigen Abschnitt wurde festgestellt, dass die Ausführungen von Anfragen mit Join-Operationen (z. B. längere Pfaden in der Anfrage) besonders kostenintensiv sein können. Die Ergebnisse in diesem Abschnitt werden darüber Aufschluss geben, ob ein Erstellen von Indexen für derartige Teile einer Anfrage und deren Verwendung in Anfragen sinnvoll sein könnte.

Dieser Abschnitt ist wie folgt gegliedert: Zunächst wird im Abschnitt 6.4.1 die Konfiguration und die Durchführung der Experimente beschrieben. Daran anschließend werden in Abschnitt 6.4.2 die Messergebnisse für diese Experimente präsentiert. Deren Bedeutung wird dann im Abschnitt 6.4.3 diskutiert und es werden Rückschlüsse auf den Ansatz der graphmuster-basierte Indexierung gezogen.

6.4.1 Konfiguration und Durchführung

Die Konfiguration und die Durchführung der Experimente wird wie zuvor in drei Teilen beschrieben. Zunächst wird auf die Definition der verwendeten Indexmuster und SPARQL-Anfragen eingegangen. Anschließend werden das Messverfahren und die Messgrößen vorgestellt und schließlich die Evaluationsumgebung dargestellt.

6.4.1.1 Datensatz, Indexmuster und SPARQL-Anfragen

Die Experimente wurden für einen ca. 100k Tripel enthaltenen Datensatz durchgeführt, der mit dem Datengenerator des Berlin SPARQL-Benchmark [BSB13] erzeugt wurde. Für diesen Datensatz wurden drei Indexe und neun Anfragen konstruiert.

Index	$\lvert\Omega_{I_k}\rvert$	KByte
I_1	658	32
I_2	6.149	344
I_3	160.829	8.880

Tabelle 6.15: Anzahl der Indexeinträge und Speicherplatzverbrauch der Tabelle

Die Graphmuster aller Indexe bestehen aus zwei Tripelmustern und wurden so gewählt, dass sie kein Sternmuster der Anfragen (s. unten) vollständig überdecken. Darüber hinaus beinhalten die Indexe unterschiedlich viele Einträge/Lösungen (vgl. Tabelle 6.15), die jeweils in einer separaten Tabelle in einer PostgreSQL-Datenbank gespeichert wurden. Ein Indexeintrag besteht dabei aus den IDs der Ressourcen und Literale, die eine Lösungsabbildung des Indexmusters darstellen. Über den Index-Tabellen wurden keine weiteren Indexe angelegt. Die Größe des durch die Indexe genutzten Sekundärspeichers kann der Tabelle 6.15 entnommen werden.

Abbildung 6.18 zeigt beispielsweise die Definition des Indexes I_1. Die Definitionen aller über den Datensatz erstellten Index finden sich in den Tabellen 6.19 und 6.20 im Anhang A.3.1.

Anfrage	$\|Q\|$	$\|\Omega\|$	$\|\Omega_{Q\setminus I_1}\|$	$\|\Omega_{Q\setminus I_2}\|$	$\|\Omega_{Q\setminus I_3}\|$
$Q_{1,1}$		14.538	6.149	658	6.149
$Q_{1,2}$	4	287	6.149	13	6.149
$Q_{1,3}$		18	6.149	1	6.149
$Q_{2,1}$		10.735	4.428	497	6.149
$Q_{2,2}$	6	242	4.428	11	4.428
$Q_{2,3}$		36	4.447	2	4.247
$Q_{3,1}$		9.527	3.952	459	6.149
$Q_{3,2}$	8	182	3.952	8	3.952
$Q_{3,3}$		18	3.952	2	3.952

Tabelle 6.16: Tripelanzahl und Ergebnisgröße der vollständigen Anfragen sowie die Ergebnisgröße ohne das jeweilige Indexmuster

Auch in diesen Experimenten war das Ziel, bestimmte Eigenschaften und nicht die allgemeine Performanz des RDF-Datenbanksystems zu untersuchen. Daher wurden für die Evaluation der graphmuster-basierten Indexierung anstelle der Anfragen eines Benchmarks manuell neun Anfragen derart konstruiert, dass sie unterschiedliche Charakteristika von Anfragen widerspiegeln. Diese Charakteristika umfassen die Anzahl der Tripelmuster und die Ergebnisgröße (vergleiche Tabelle 6.16). Die Anfragen sind entsprechend ihrer Größe in drei Gruppen aufgeteilt: $Q_{1,n}$ mit 4, $Q_{2,n}$ mit 6 und $Q_{3,n}$ mit 8 Tripelmustern. Ein Anfragemuster mit einer Größe kleiner als 4 ist nicht sinnvoll, da die Indexmuster selbst schon zwei Tripelmuster groß sind. Tabelle 6.16 listet die Ergebnisgrößen der Anfragemuster auf, wenn von dem Basis-Graphmuster der Anfrage $Q_{j,j}$ das Indexmuster I_k entfernt wurde.

Die Basis aller Anfragen bildet ein Sternmuster bestehend aus drei Tripelmustern und einem einzelnen Tripelmuster, das mit einem Objekt des Sternmusters verknüpft ist. Wie die Ergebnisse im vorherigen Abschnitt gezeigt haben, können sternförmige Anfragen mit konstantem Subjekt effizient ausgeführt werden und die Verwendung eines Indexes wäre nicht sinnvoll. Daher wurde in den nachfolgenden Experimenten keine Sternmuster mit konstanten Subjekten untersucht.

Bei der Verwendung eines Indexes treten Abbildungen zwischen Indexmuster und Anfrage auf, bei denen eine Variable des Index an einen konstanten Wert des Anfragemusters gebunden wird. Dies hat zur Folge, dass die Einträge in der relationalen Datenbank gefiltert werden müssen. Tabelle 6.17 gibt einen Überblick, welche Anfragen eine Selektion auf der Indextabelle erfordern.

```
select * where {
   ?x  v:productFeature ?u .
   ?u  d:date ?q .
}
```

Abbildung 6.18: Der Definition des Indexes I_1 zugrunde liegende Anfrage

	$Q_{1,1}$	$Q_{1,2}$	$Q_{1,3}$	$Q_{2,1}$	$Q_{2,2}$	$Q_{2,3}$	$Q_{3,1}$	$Q_{3,2}$	$Q_{3,3}$
I_1		×	×		×	×		×	×
I_2									
I_3	×	××	××	×	××	××	×	××	××

Tabelle 6.17: × = eine Variable des Indexmusters wird an eine Konstante der Anfrage gebunden (×× = zwei Variablen)

Damit die Anfragen unterschiedlich große Ergebnismengen zurückliefern und somit verschiedene Szenarien abgedeckt werden können, wurden geeignete Objekte der Tripelmuster ausgewählt. Dabei wurde darauf geachtet, dass die Anfragen $Q_{i,1}$, $Q_{i,2}$ und $Q_{i,3}$ jeweils Ergebnisse in der gleichen Größenordnung zurückliefern (vgl. Tabelle 6.16). Abbildung 6.19 zeigt als Beispiel die Anfrage $Q_{3,2}$; eine vollständige Auflistung der Definitionen aller Indexe und Anfragen findet sich im Anhang A.3. Das Sternmuster mit dem Subjekt `?x` ist allen Anfragen gemein, sie unterscheiden sich nur im Objekt des zweiten Tripelmusters – hier `i:ProductFeature24`. Die beiden anderen Sternmuster haben die Subjekte `?r1` und `?r2`.

```
select * where {
   ?x  v:productFeature ?u ;
       v:productFeature i:ProductFeature24 ;
       v:productFeature i:ProductFeature1 .
   ?u  d:date ?q .
   ?r1 v:reviewFor ?x ;
       v:rating3 10 .
   ?r2 v:reviewFor ?x ;
       v:rating3 9 .
}
```

Abbildung 6.19: Definition der Anfrage $Q_{3,2}$

6.4.1.2 Messgrößen und Messverfahren

Zum Evaluieren des Einflusses des graphmuster-basierten Indexierens auf die Ausführungszeit werden die Anfragen in einen vom Index überdeckten und den Rest zerlegt. Die Lösungen des Indexes und die des restlichen Anfragemusters werden über eine Join-Operation miteinander verknüpft. Das für die Experimente verwendete Anfragesystem ARQ nutzt für die Berechnung des Joins einen Nested-Loop-Join, wobei die Lösungen des rechten Operands vollständig in den Hauptspeicher geladen werden. Anschließend wird für jede Lösung des anderen Operands alle Lösungen des rechten durchlaufen. Diese Implementierung hat den Nachteil, dass bei zu großem Zwischenergebniss des rechten Operanden das Ergebnis des Joins aufgrund unzureichenden Hauptspeichers nicht bzw. nicht in vertretbarem Zeitrahmen berechnet werden kann. Aus diesem Grund konnten die Experimente nur mit einem 100k umfassenden Datensatz durchgeführt werden.

Als Vergleichssystem wurde für die Experimente erneut Jena TDB herangezogen. Der Optimierer von Jena TDB generiert Anfragepläne, die nicht die Join-Operation von ARQ sondern eine TDB-spezifische Implementierung verwenden. Da der im ARQ verwendete Nested-Loop-Join zu einer erheblichen Erhöhung der Ausführungszeit führt, wäre ein direkter Vergleich der beiden Systeme nicht möglich. Vorbereitende Experimente haben gezeigt, dass die Ausführungszeiten unter Verwendung des nicht modifizierten TDB-Systems um Faktor 16-20 geringer sind (vgl. Tabellen 6.18, 6.19 und 6.20). Deshalb wurden die von Jena TDB generierten Ausführungspläne so modifiziert, dass auch in den Fällen ohne Verwendung eines Indexes die Anfrage zerlegt und eine Join-Operation eingefügt wird.

	$Q_{1,1}$	$Q_{1,2}$	$Q_{1,3}$	$Q_{2,1}$	$Q_{2,2}$	$Q_{2,3}$	$Q_{3,1}$	$Q_{3,2}$	$Q_{3,3}$
TDB	318	7	4	499	16	5	777	18	5

Tabelle 6.18: Mittlere Ausführungszeiten (in ms) der TDB ohne Modifikation des Ausführungsplanes

Abbildung 6.20 veranschaulicht dieses Vorgehen anhand eines Beispiels. Die ursprüngliche Anfrage besteht aus einem einzelnen Basis-Graphmuster BGP. Falls der Optimierer sich für die Verwendung des Indexes I_1 entscheidet, so wird das Graphmuster BGP in die Teilanfragen BGP_1 und BGP_2 zerlegt. Um die Lösungen für BGP_2 zu ermitteln, wird ein Indexzugriff op_{I_1} verwendet (linker Ausführungsplan in Abbildung 6.20). Um im Falle einer Ausführung ohne Indexzugriff einen möglichst ähnlichen Ausführungsplan zu erhalten, wird ein Ausführungsplan erzeugt, wie er auf der rechten Seite zu sehen ist. Damit muss in jedem Falle dieselbe Join-Operation ausgeführt werden und der darin verwendete Algorithmus fällt somit nicht ins Gewicht.

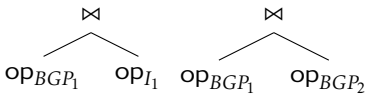


Abbildung 6.20: Ausführungspläne mit Indexzugriff (links) und ohne (rechts)

Alle Anfragen werden zum einen unter Verwendung einer der Indexe I_1, I_2 und I_3 und zum anderen ohne Index aber mit identischer Zerlegung der Anfrage (s. oben) ausgeführt. Darüber hinaus wird für jede Anfrage der Ausführungsplan dahingehend modifiziert, dass der Indexzugriff einmal der linke Operand und einmal der rechte Operand in der Join-Operation ist. Hierdurch kann der Einfluss des Ladens der Ergebnisses des rechten Operands in den Hauptspeicher untersucht werden.

Damit die Caches des Datenbanksystems gefüllt sind, wird vor jedem Durchlauf eine Anfrage an das Datenbanksystem gestellt, deren Ausführungszeiten in der Evaluation nicht berücksichtigt wird. Des Weiteren wird jede Anfrage insgesamt zehn Mal nacheinander berechnet und der Mittelwert der resultierenden Messwerte als Ausführungszeit für eine Anfrage angenommen. Damit das Datenbanksystem auch alle Lösungen zu einer Anfrage materialisieren muss, werden alle Ergebnisse der Anfrage abgerufen. Die gemessenen Ausführungszeiten umfassen somit die Zeit für das Parsen, Optimieren und Ausführen einer Anfrage sowie das Verarbeiten der Lösungen. Da der Aufwand

für das Generieren der Ergebnisse unabhängig vom Indexzugriff ist – die Ergebnisgröße verändert sich nicht –, kann dieser im Folgenden vernachlässigt werden.

Für die Evaluation des Ansatzes werden die absoluten Ausführungszeiten der Anfragen mit und ohne Indexzugriff ins Verhältnis gesetzt. Im Folgenden wird dieses Verhältnis als Faktor f bezeichnet[3]. Die Verwendung eines Faktors ist ein unmittelbarer Indikator für einen Leistungsgewinn bzw. -verlust. Die Werte eines Faktors haben dabei die folgenden Bedeutungen:

$$f < 1 \Rightarrow \text{Jena TDB schneller als RCS}$$
$$f = 1 \Rightarrow \text{beide System gleich}$$
$$f > 1 \Rightarrow \text{Jena TDB langsamer als RCS}$$

6.4.1.3 Evaluationsumgebung

Die Experimente wurden auf einem Server unter dem Betriebssystem Solaris 10 (64 Bit). Der Server verfügt über vier CPUs vom Typ Dual-Core AMD Opteron und 32 GB Hauptspeicher. Zum Ausführen der Software wurde Java in der Version 1.7.0_21 verwendet. Die verwendeten Software-Komponenten sind zusammen mit der jeweilig verwendeten Version in Tabelle 6.1 auf Seite 140 aufgelistet.

6.4.2 Ergebnisse und Beobachtungen

Im Folgenden werden die Messergebnisse der durchgeführten Experimente dargestellt und Beobachtungen zu dem Verhalten der Ausführungszeiten aufgelistet. Zunächst werden die Ergebnisse und Beobachtungen von den Ausführungsplänen präsentiert, in denen der Indexzugriff der rechte Operand des Join-Operators war. Anschließend wird der Fall betrachtet, beim dem der Indexzugriff der linke Operand ist. Ein Zusammenführen der Beobachtungen und die Diskussion der Ergebnisse erfolgt im nachfolgenden Abschnitt 6.4.3.

6.4.2.1 Indexzugriff als rechter Join-Operand

Die Diagramme in den Abbildungen 6.21 bis 6.23 zeigen die mittleren Ausführungszeiten für das Ausführen der Anfragen mit und ohne Verwendung eines Indexes, wobei der Index der rechte Operand des Joins ist. In einem Diagramm werden jeweils die Messwerte für die Anfragen dargestellt, die ungefähr gleich viele Ergebnisse zurückliefern (vgl. Tabelle 6.16 auf Seite 167). Somit gibt es für die Anfragegruppen $Q_{i,1}$, $Q_{i,2}$ und $Q_{i,3}$ jeweils ein Diagramm. Die den Diagrammen zugrunde liegenden Messwerte finden sich im Tabelle 6.19.

Aus den Diagrammen ergeben sich die folgenden Beobachtungen:

Beobachtung 6.24. Abbildung 6.21 zeigt die Messwerte für die Anfragen mit den größten Ergebnissen. Aus dem Diagramm ist ersichtlich, dass die Ausführung mit Indexzugriff bis auf die Kombination I_1 und $Q_{1,1}$ schneller ist als ohne. Der Faktor zwischen der Ausführung ohne und mit Index hängt dabei

[3] Im Folgenden wird auch die Bezeichnung *Leistungsgewinn* bzw. *Leistungsverlust* verwendet, um die Reduktion bzw. Erhöhung der Ausführungszeit durch das Verwenden von eines Indexes auszudrücken.

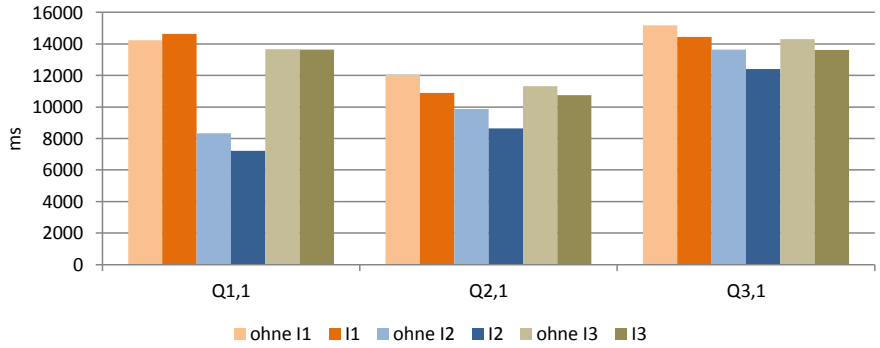


Abbildung 6.21: Mittlere Ausführungszeiten der Anfragen $Q_{i,1}$ mit und ohne Indexzugriff (als rechter Join-Operand)

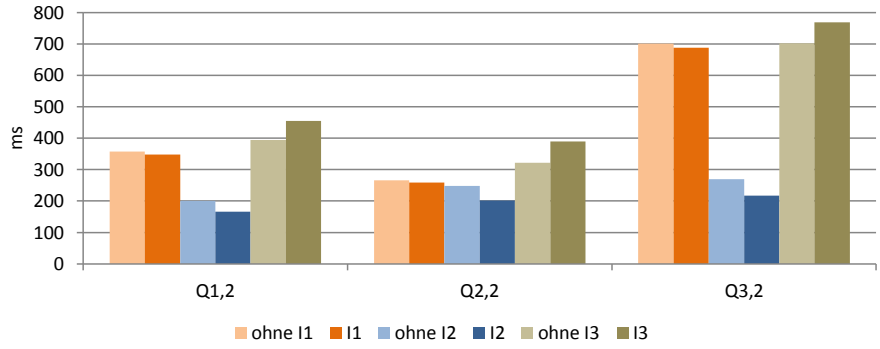


Abbildung 6.22: Mittlere Ausführungszeiten der Anfragen $Q_{i,2}$ mit und ohne Indexzugriff (als rechter Join-Operand)

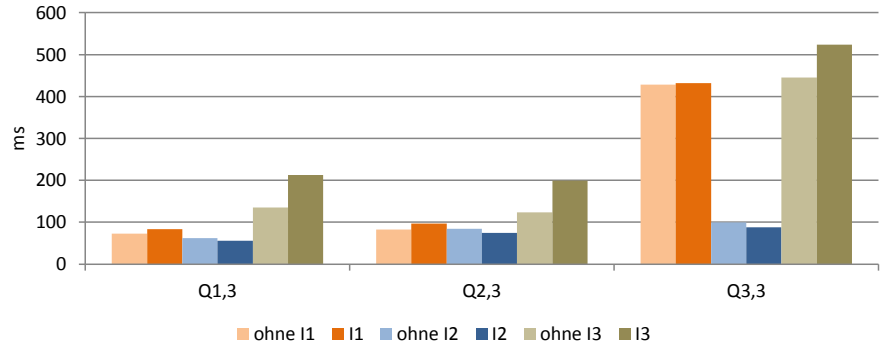


Abbildung 6.23: Mittlere Ausführungszeiten der Anfragen $Q_{i,3}$ mit und ohne Indexzugriff (als rechter Join-Operand)

	ohne I_1	I_1	Faktor	ohne I_2	I_2	Faktor	ohne I_3	I_3	Faktor
$Q_{1,1}$	14239	14645	0,97	8341	7207	1,16	13668	13648	1,00
$Q_{2,1}$	12022	10883	1,10	9865	8639	1,14	11311	10755	1,05
$Q_{3,1}$	15167	14446	1,05	13629	12399	1,10	14307	13606	1,05
$Q_{1,2}$	357	348	1,03	200	166	1,21	393	455	0,86
$Q_{2,2}$	266	259	1,03	247	202	1,23	322	389	0,83
$Q_{3,2}$	701	688	1,02	269	217	1,24	702	768	0,91
$Q_{1,3}$	73	83	0,88	61	55	1,11	135	212	0,63
$Q_{2,3}$	82	96	0,85	84	74	1,14	123	198	0,62
$Q_{3,3}$	428	431	0,99	99	88	1,13	445	523	0,85

Tabelle 6.19: Mittlere Ausführungszeiten (in ms) mit/ohne Indexe; Indexzugriff ist *rechter* Operand des Join-Operators

von dem verwendeten Index ab und bewegt sich zwischen 1,00 und 1,16. Vergleicht man den Leistungsgewinn über alle Anfragen $Q_{i,1}$ hinweg, so wurde mit der Verwendung des Index I_2 immer die größte Verbesserung erzielt.

Bei den Anfragen mit den mittleren Ergebnisgrößen, Abbildung 6.22, zeigt grundsätzlich ein ähnliches Verhalten der Ausführungszeiten. Nur bei Verwenden des Indexes I_3 ergibt sich mit über alle Anfragen hinweg eine schlechtere Ausführungszeit als ohne Indexzugriff. Der Faktoren für die Indexe I_1 und I_2 bewegen sich zwischen 1,02 und 1,24 und für den Index I_3 zwischen 0,86 und 0,91.

Im Falle der Anfragegruppe mit den kleinsten Ergebnisgrößen konnten nur verbesserte Ausführungszeiten unter Verwendung des Indexes I_2 erzielt werden (Faktor von ca. 1,12). Die anderen beiden Indexe zeigen ein schlechteres Verhalten als ohne Indexzugriff. Die Faktoren hierbei betragen durchschnittlich 0,91 für den Index I_1 und 0,70 für I_3. □

Beobachtung 6.25. In fünf Fällen sinkt der Leistungsgewinn, wenn die Anfragen ein kleineres Ergebnis haben (z. B. $Q_{2,1}$, $Q_{2,2}$ und $Q_{2,3}$). In allen anderen Fällen ist der Faktor bei den Anfragen mit mittlerem Ergebnis am höchsten. In allen Experimenten ist jedoch der Leistungsgewinn bei den Anfragen mit dem kleinsten Ergebnis am geringsten. □

Beobachtung 6.26. Betrachtet man die Faktoren je Index über alle Anfragen hinweg, so ist der Leistungsgewinn bei Verwendung von Index I_2 (Durchschnitt 1,16) deutlich höher als bei den Indexen I_1 (0,99) und I_3 (1,02). □

Beobachtung 6.27. Vergleicht man die Faktoren für die jeweiligen Gruppen von wachsende Anfragemuster (z. B. $Q_{1,3}$, $Q_{1,2}$ und $Q_{1,1}$), dann kann keine eindeutige Aussage über das Verhalten des Faktors getroffen werden. Bei einigen Gruppen (z. B. $Q_{i,3}$ und Index I_2) wird ein besserer Faktor mit einem großen und bei anderen (z. B. $Q_{i,1}$ und Index I_2) mit einem kleinen Anfragemuster erzielt. In vier Fällen ist der Leistungsgewinn beim größten Anfragemuster am höchsten. Für die mittlere Anfragegröße gibt es nur zwei und für die kleinste genau einen Fall. In zwei weiteren Fällen gibt es keine eindeutige Spitze. □

6.4.2.2 Indexzugriff als linker Join-Operand

Die Abbildungen 6.24 bis 6.26 geben die Ausführungszeiten für die Anfragepläne wieder, in denen der Indexzugriff der linke Operand der Joins war. Auch

diese Diagramme zeigen die Ausführungszeiten mit und ohne Indexzugriffe; für jede Anfragegruppen $Q_{i,1}$, $Q_{i,2}$ und $Q_{i,3}$ ein Diagramm. Die den Diagrammen zugrunde liegenden Messwerte finden sich im Tabelle 6.20.

	ohne I_1	I_1	Faktor	ohne I_2	I_2	Faktor	ohne I_3	I_3	Faktor
$Q_{1,1}$	8084	5483	1,47	13822	3978	3,47	8078	5299	1,52
$Q_{2,1}$	5657	3791	1,49	10698	3485	3,07	5561	3830	1,45
$Q_{3,1}$	6044	4546	1,33	10086	3203	3,15	6067	4661	1,30
$Q_{1,2}$	173	147	1,17	317	129	2,45	233	282	0,83
$Q_{2,2}$	129	112	1,15	288	123	2,34	193	246	0,78
$Q_{3,2}$	496	480	1,03	223	105	2,11	522	598	0,87
$Q_{1,3}$	56	58	0,96	73	74	1,00	118	195	0,61
$Q_{2,3}$	46	48	0,95	98	61	1,60	109	186	0,59
$Q_{3,3}$	412	414	1,00	96	62	1,56	443	519	0,85

Tabelle 6.20: Mittlere Ausführungszeiten (in ms) mit/ohne Indexe; Indexzugriff ist *linker* Operand des Joins-Operators

Anhand der Diagrammen können die folgenden Beobachtungen gemacht werden:

Beobachtung 6.28. Das erste Diagramm in Abbildung 6.24 zeigt für alle Anfragen eine geringere Ausführungszeit bei Verwendung der Indexe. Im Falle von Index I_2 verringern sich die Ausführungszeiten um den Faktor 3,07 bis 3,47. Mit den beiden Indexen I_1 und I_3 erreichen einen Faktor zwischen 1,30 und 1,52.

Bei den Anfragen mit der mittleren Ergebnisgröße beschleunigt das Verwenden der Indexe I_1 und I_2 die Anfrageausführung um durchschnittlich Faktor 1,12 bzw. 2,3. Ein Indexzugriff auf I_3 wirkt sich dahingegen negativ auf die Ausführungszeit aus (durchschnittlicher Faktor 0.83).

In der letzten Anfragegruppe mit der kleinen Ergebnisgröße wurde eine Verbesserung der Ausführungszeiten nur für die Anfrage-Index-Kombinationen $Q_{2,3}$, $Q_{3,3}$ und I_2 erzielt (Faktoren 1,60 und 1,56). Der Index I_3 wirkt sich mit einem durchschnittlichen Faktor von 0,68 wiederum negativ auf die Ausführungszeit aus. Bei den restlichen Anfragen ergibt sich nur ein geringfügiger Unterschied zwischen den beiden Systemen. □

Beobachtung 6.29. Betrachtet man den Leistungsgewinn für Anfragegruppen mit einem kleiner werdenden Ergebnis (z. B. $Q_{1,1}$, $Q_{1,2}$ und $Q_{1,3}$), dann ist der Leistungsgewinn in all Experimenten bei den Anfragen mit einem großen Ergebnis am höchsten. □

Beobachtung 6.30. Wenn man je Index die Faktoren über alle Anfragen hinweg betrachtet, so wirken sich die beiden Indexe I_2 und I_1 positiv auf die Ausführungszeit aus (durchschnittlicher Faktor 2,31 bzw. 1,17). Mit dem Index I_3 konnte nur ein durchschnittlicher Faktor von 0,98 erreicht werden. □

Beobachtung 6.31. Für die jeweiligen Gruppen von Anfragemuster mit einer wachsenden Anzahl an Tripelmustern (z. B. $Q_{1,1}$, $Q_{2,1}$ und $Q_{3,1}$) kann keine Korrelation beobachtet werden. In vier Fällen ist der Leistungsgewinn beim kleinsten, in zwei Fällen beim mittleren und in drei Fällen beim größten Anfragemuster am höchsten. Nur in drei Fällen sinkt der Leistungsgewinn mit wachsendem Anfragemuster. □

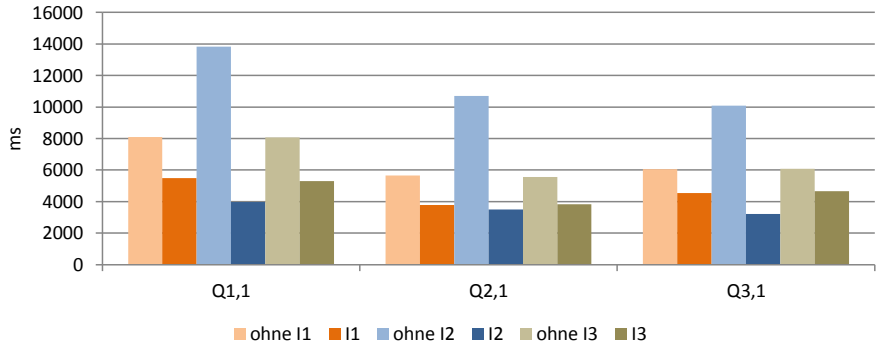


Abbildung 6.24: Mittlere Ausführungszeiten der Anfragen $Q_{i,1}$ mit und ohne Indexzugriff (als linker Operand)

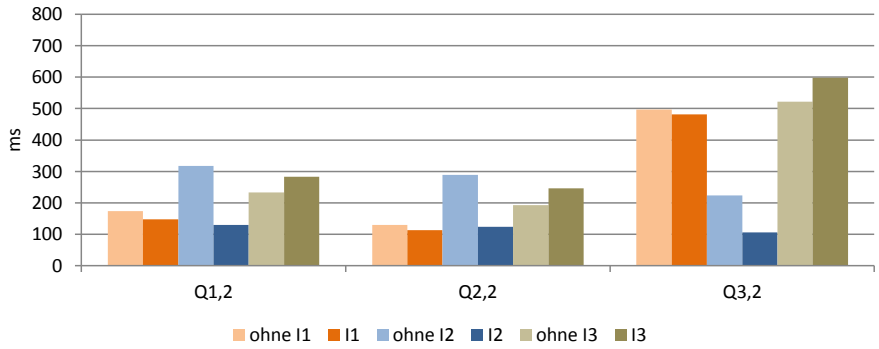


Abbildung 6.25: Mittlere Ausführungszeiten der Anfragen $Q_{i,2}$ mit und ohne Indexzugriff (als linker Operand)

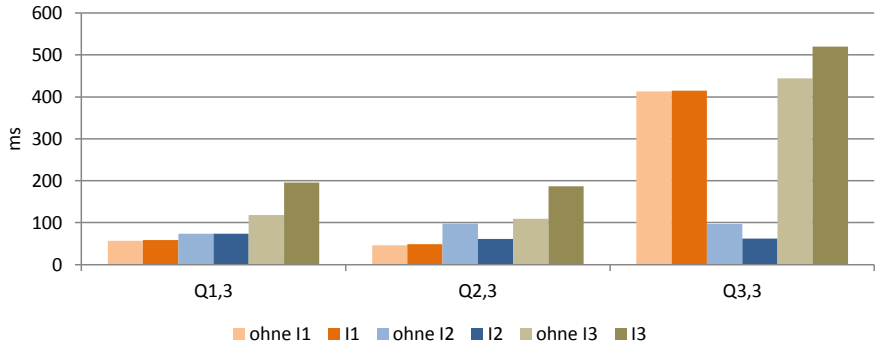


Abbildung 6.26: Mittlere Ausführungszeiten der Anfragen $Q_{i,3}$ mit und ohne Indexzugriff (als linker Operand)

6.4.3 Diskussion und Schlussfolgerungen

Die im vorherigen Abschnitt präsentierten Ergebnisse werden im Folgenden zusammengeführt und diskutiert. Dabei wird insbesondere auf die Einflüsse der Position des Indexzugriffs im Ausführungsplan, der Ergebnisgröße, der Anfragegröße und der Indexgröße eingegangen.

Linker vs. rechter Operand. Zunächst konnte Anhand der Experimente festgestellt werden, dass die Position des Indexzugriffs im Ausführungsplan (linker vs. rechter Join-Operand) erheblichen Einfluss auf den Leistungsgewinn hat. Für nahezu alle Anfrage-Index-Kombinationen konnten höhere Faktoren erreicht werden, wenn der Indexzugriff linker Join-Operand war. Dieses Verhalten wird insbesondere durch den Index I_2 veranschaulicht (Beobachtungen 6.26 und 6.30).

Die Ergebnisse des rechten Operands werden durch das Anfragesystem ARQ in den Hauptspeicher geladen, während über die Ergebnisse des linken iteriert wird. Daher kann angenommen werden, dass das Bestimmen der Lösungen für das überdeckte Graphmuster durch Jena TDB aufwändiger als das Lesen einer Indextabelle durch PostgreSQL. Nur wenn die Größe des Indexes deutlich größer als die Ergebnisgröße des nicht-überdeckten Anfragemusters ist (z. B. Index I_3 mit ca. 160k Einträgen), wird das überdeckte Anfragemuster performanter durch Jena TDB ausgeführt.

Diese Erkenntnisse legen den Schluss nahe, dass die in dieser Arbeit vorgeschlagene Systemarchitektur für das Erstellen und Verwalten von Indexen über Graphmuster grundsätzlich sinnvoll ist. Jedoch zeigt das recht unterschiedliche Verhalten bei den Indexzugriffen als linker oder rechter Join-Operand und auch die Ausführungszeiten der unmodifizierten Jena-TDB, dass die Implementierung des Join-Algorithmus wesentlichen Einfluss auf die Anfrageperformanz hat.

Einfluss der Ergebnisgröße. Unabhängig von der Position des Indexzugriffs im Ausführungsplan kann eine Korrelation zwischen dem erzielten Leistungsgewinn und der Größe des Ergebnis festgestellt werden: großes Ergebnis $\Rightarrow$ hoher Faktor (vgl. Beobachtungen 6.25 und 6.29). Dies wird insbesondere durch die Experimente untermauert, in denen der Indexzugriff der linke Join-Operand ist.

Nimmt man den Fakt hinzu, dass es bei der Größe die Zwischenergebnisse des nicht-überdeckten Anfragemusters für die Indexe I_1 und I_3 in fast allen Fällen nur geringfügige Unterschiede gibt (vgl. Tabelle 6.16 auf Seite 6.16), dann muss im Fall Auswertung ohne Indexzugriff die Berechnung der Lösungen für den überdeckten Teil der Anfrage einen wesentlichen Anteil einnehmen. Als Konsequenz kann geschlussfolgert werden, dass das Verwenden von Index-Operatoren in den Ausführungsplänen insbesondere dann sinnvoll ist, wenn große Ergebnisse prognostiziert werden.

Einfluss der Anfragegröße. Aus den Beobachtungen 6.27 und 6.31 zeigen uneinheitlichen Leistungsgewinn unter Veränderung der Anfragegröße. Daraus kann geschlussfolgert werden, dass die Größe des Anfragemusters nur eine untergeordnete Rolle für den Leistungsgewinn im Zusammenhang mit der Verwendung von Index spielt.

Einfluss der Indexgröße. In den Ergebnissen der Experimente konnte beobachtet werden, dass von allen Indexen der Index I_2 den größten Leistungsgewinn erreichte (Beobachtungen 6.26 und 6.30). Da der Index I_1 weniger und I_3 mehr Einträge enthalten, kann die Indexgröße dieses Verhalten nicht erklären. Vergleicht man die Indexmuster aller Indexe (vgl. Anhang A.3.1 auf Seite 189), dann haben die beiden Indexe I_1 und I_3 untereinander eine recht ähnliche aber zu Index I_2 verschiedene Struktur. Die ersten Indexe sind über einem sternförmigen Graphmuster definiert, jedoch hat das Muster von I_1 einen konstanten Wert anstelle einer Variable. Im Gegensatz dazu ist das Graphmuster von Index I_2 ein aus zwei Tripeln bestehender Pfad.

In den Anfragen Qi, j überdecken die beiden Indexmuster I_1 und I_3 einen Teil eines sternförmiges Graphmusters. Das Indexmuster von I_2 hingegen überdeckt einen Pfad in den Anfragemuster. Die Beobachtungen 6.26 und 6.30 bestätigen einen positiver Einfluss des Verwendens von Index I_2 auf den Leistungsgewinn für alle Anfragen. Des Weiteren kann festgestellt werden, dass Jena TDB aufgrund seines Speichermodells insbesondere sternförmige Anfragen performant ausführen kann (vgl. Abschnitt 3.1.2).

Eine wesentliche Schlussfolgerung aus diesen Beobachtungen ist, dass die Kombination von Indexen über Pfaden und einem auf die Auswertung von sternförmigen Anfragen spezialisierten RDF-Datenbanksystem zu einer Verringerung auf die Ausführungszeit von Anfragen auswirkt.

6.5 Zusammenfassung

In dem Kapitel wurden die Ergebnisse der Evaluation der in dieser Arbeit entwickelten Ansätze zum Verwalten und Indexieren von RDF-Daten präsentiert und diskutiert.

Die Evaluation von sternförmigen Anfragen und Join-Operation zwischen diesen hat gezeigt, dass das im Abschnitt 3.2.1 theoretisch beschriebene Verhalten des RCS-Speichermodells im Wesentlichen auch in der prototypischen Implementierung beobachtet werden konnte. Sternförmige Anfragen werden mit konstanter Komplexität beantwortet und bei Join-Anfragen verhält sich der RCS grundsätzlich ähnlich zu relationalen Datenbanksystemen.

Die Diskussion der Ergebnisse offenbarte jedoch auch einen nicht zu vernachlässigbaren Einfluss der dem Prototyp zugrunde liegenden Software-Komponenten. Zum einen wurde in der Evaluation festgestellt, dass nur ca. 20 % des belegten Speicherplatzes durch die Datenseiten und die Werte der Zugriffsstrukturen eingenommen wird – der restliche Speicherplatz wurde durch die zum Speichern der Daten verwendeten Berkeley DB eingenommen. Zum anderen begründen die Ergebnisse die Vermutung, dass die Anfrageverarbeitungskomponente ARQ des Jena2-RDF Frameworks einen spürbaren Einfluss auf die Ausführungszeit hat, wenn entweder Anfragen mit vielen Tripelmustern oder Join-Operationen involviert sind. Beide Beobachtungen legen nahe, dass eine erneute Evaluation mit einer auf das RCS abgestimmten Datenverwaltungs- und Anfrageverarbeitungskomponente durchzuführen werden sollte.

Bei der Evaluation der Indexierung von Basis-Graphmuster konnte kein Zusammenhang zwischen Größe des Anfragemusters und dem Leistungsgewinn festgestellt werden. Vielmehr scheinen die Größe des Anfrageergebnis-

ses und die Struktur des Indexmusters einen Einfluss auf den Leistungsgewinn zu haben. In den Experimenten konnte ein höherer Leistungsgewinn bei den Anfragen mit einem großen Endergebnis und unter Verwendung eines Indexes mit einem pfadartigen Basis-Graphmuster erzielt werden.

Auch in diesen Experimenten konnte der Einfluss des Jena2-RDF-Frameworks nachgewiesen werden. Insbesondere kam zum Vorschein, dass die Ausführungszeit stark von der Position des Indexzugriffs im Anfrageausführungsplan abhängt – in der Position als linker Join-Operand wurden geringere Ausführungszeiten gemessen. Aus dieser Perspektive erscheint es ebenfalls sinnvoll, ein speziell auf die Verwendung von Indexen abgestimmtes Systemarchitektur zu entwickeln.

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und anschließend ein Ausblick auf mögliche Verbesserungen für die Verwaltung und Indexierung von RDF-Daten gegeben.

7.1 Zusammenfassung

In dieser Arbeit wir der Resource Centered Store (RCS) zum Verwalten von RDF-Daten präsentiert. Mit dem RCS wurde ein neuartiges Speichermodell definiert (vgl. Kapitel 3), dessen wesentliche Eigenschaft die Gruppierung der Tripel anhand ihres Subjekts ist. Im Gegensatz zu existierenden nativen RDF-Datenbankmanagementsystemen konnte im RCS eine an die relationalen DBMS angelehnte, auf Datenseiten basierende Datenverwaltung. Somit ermöglicht der RCS ein Übertragen von relationalen Algorithmen und Zugriffsstrukturen auf ein natives RDF-Datenbanksystem.

Als Konsequenz des RCS-Speichermodells können sternförmige Anfragen, unabhängig von der Größe ihres Graphmusters, ohne eine einzige Join-Operation beantwortet werden. Aufgrund dieser Eigenschaft konnten Anfragen selbst mit der prototypischen Implementierung des RCS zeiteffizienter als vom etablierten, nativen RDF-Repository Jena2 TDB beantwortet werden. Zur Auswertung einer beliebigen SPARQL-Anfrage wird diese in sternförmige Teilanfragen zerlegt und deren Ergebnisse miteinander verknüpft (Join-Operation). Um den Aufwand zur Berechnung dieser Join-Operationen zu verringern, wurde in Kapitel 4 ein Konzept zum graphmuster-basierten Indexieren von RDF-Daten beschrieben. In der Evaluation konnte gezeigt werden, dass ein Indexzugriff bei der Berechnung von SPARQL-Anfragen mit Pfaden besonders geeignet ist.

In Kapitel 5 wurde das Generieren von Ausführungsplänen von SPARQL-Anfragen untersucht. Aufbauend auf dem SPARQL Query Graph Model wurden Operationen und Transformationsregeln definiert, die das Ausnutzen der Eigenschaften des RCS-Speichermodells erlauben. Mit Hilfe der Transformationsregeln kann beispielsweise eine Anfrage dahingehend verändert werden,

dass entweder große sternförmige Graphmuster entstehen oder das Verwenden eines Indexes ermöglicht wird.

In den Experimenten mit der prototypischen Implementierung des RCS konnten die theoretisch erwarteten positiven Eigenschaften bezüglich der Auswertung belegt werden. Für die graphmuster-basierte Indexierung konnten Anfragetypen identifiziert werden, für die sich ein Indexzugriff günstig auf die Anfrageperformanz auswirkt.

7.2 Zukünftige Arbeiten

Im Laufe der Entwicklung und Evaluation des RCS-Speichermodells und der Indexierung konnten einige Möglichkeiten identifiziert, die zum einen eine Verbesserung des jeweiligen Konzepts darstellen oder deren detailliertere Evaluation ermöglichen.

Indexierung von Prädikaten und Objekten Mit den im RCS-Speichermodell vorgesehenen Zugriffsstrukturen für Prädikate und Objekte konnte während der Anfrageausführung nicht festgestellt werden, ob dasselbe Tripel ein bestimmtes Prädikat und ein bestimmtes Objekt beinhalteten. Es konnte nur ermittelt werden, welches Subjekt mit dem jeweiligen Prädikat bzw. Objekt in Beziehung steht. Der Hauptgrund für diese Design-Entscheidung war die Vermutung, dass die resultierenden Bitsets nicht im Hauptspeicher gehalten werden könnten – insbesondere, da für jedes distinkte Prädikat und Objekt ein Bitset angelegt wird.

Die Evaluation ergab, dass die Bitsets in den beiden Indexen nur dünn besetzt waren und damit der Speicherbedarf gering war. Daher könnte in zukünftigen Arbeiten untersucht werden, ob in den Bitsets für jedes Tripel – anstelle von jedem Subjekt – ein Bit vorgesehen werden kann. Auf diese Weise könnte in der Anfragebearbeitung genauer die infrage kommenden Datenseiten ermittelt werden.

Bei der Indexierung von Objekten wurden Literale in dieser Arbeit nicht weiter betrachtet. Weiterführende Arbeiten könnten geeignete Zugriffsstrukturen zum Lokalisieren von Literalen untersuchen, z. B. Bereichsanfragen oder reguläre Ausdrücke.

Indexverwaltung Die in der Arbeit vorgestellte Indexierung basiert auf dem Materialisieren der Lösungsabbildungen für ein Basis-Graphmuster. Die Evaluation deutet darauf hin, dass sternförmige Indexmuster einen geringeren Performanzgewinn bei der Anfrageausführung erzielen

Weiterführende Arbeiten könnten untersuchen, ob im Falle von sternförmigen Indexmustern ein direkter Verweis auf das betreffenden Subjekt in den Datenseiten eine effizientere Auswertung von Graphmustern ermöglichen kann. An diesem Ansatz könnte vorteilhaft sein, dass die referenzierten Subjekte als Ausgangspunkt für eine weitergehende Auswertung des Anfragemusters – anstelle der jetzt erforderlichen Join-Operation – genutzt werden könnten. Als weiterer Aspekt könnte untersucht werden, inwiefern ein Indexieren von sternförmigen Graphmustern bei Vorhandensein der zuvor beschriebenen, tripelbasierten Prädikat- und Objektindexe überhaupt noch sinnvoll ist. Unter Umständen wäre ein Indexieren von häufig angefragten Pfaden ausreichend.

Systemarchitektur In Kapitel 6 wurde die prototypische Implementierung des RCS-Speichermodells und der graphmuster-basierten Indexierung präsentiert. Die Evaluation des Prototyps verdeutlichte, dass die verwendeten Software-Bibliotheken Berkeley-DB und Jena2-RDF-Framework einen spürbaren Einfluss auf die Ergebnisse hatten.

Die Berkeley-DB wurde im RCS zur Verwaltung von Datenseiten und Zugriffsstrukturen verwendet. Die Beobachtungen in Abschnitt 6.3.2 haben einen hohen Speicherplatzbedarf aufgezeigt. In der Arbeit wurde die begründete Vermutung aufgestellt, dass ein speziell auf das RCS abgestimmte Datenverwaltung die Anfrageperformanz positiv beeinflussen wird. Die in Jena implementierten Algorithmen sind auf die Evaluation von einzelnen Tripelmustern ausgerichtet und unterstützen nur unzureichend die Auswertung von sternförmigen Graphmustern. Eine erneute Evaluation mit verbesserten Software-Komponenten könnte einen tiefer gehenden Einblick in das Anfrageausführungsverhalten des RCS-Datenmodells geben. Darüber hinaus könnten alternative Join-Strategien evaluiert werden.

Spezialisierte Join-Algorithmen Der Fokus dieser Arbeit lag auf der Validierung des RCS-Speichermodells als geeigneter Ansatz für die Verwaltung von RDF-Daten. Da dessen Art und Weise der Datenhaltung stark dem von relationalen Datenbankmanagementsystem entspricht, kann die Hypothese aufgestellt werden, dass die relationalen Join-Algorithmen und Zugriffsstrukturen nahezu unverändert auf den RCS übertragen werden können. Diese Hypothese müsste in zukünftigen Arbeiten untersucht werden.

SPARQL Version 1.1 In der SPARQL Version 1.1 [PS13] wurden zu regulären Ausdrücken ähnliche Notationen eingeführt, um einen Pfad innerhalb eines RDF-Graphen kompakt beschreiben zu können. Beispielsweise referenziert der Ausdruck `?person foaf:knows+ ?other` alle Pfade mit einer Länge größer Null zwischen zwei Ressourcen, die nur die Eigenschaft `foaf:knows` beinhalten. Als Teil von zukünftigen Arbeiten bleibt zu untersuchen, inwiefern das RCS-Speichermodell die Auswertung von Pfadausdrücken unterstützen kann.

Anhang A

Anfragen und Indexe

In diesem Kapitel werden die für die Evaluation des RCS-Speichermodells und der graphmuster-basierten Indexierung verwendeten Anfragen aufgelistet.

Die Namensräume sind wie folgt definiert:

d	http://purl.org/dc/elements/1.1/
s	http://purl.org/stuff/rev#
v	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/
i	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
r	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromRatingSite1/
p	http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer11/
rdfs	http://www.w3.org/1999/02/22-rdf-syntax-ns#

A.1 Evaluation des RCS-Speichermodells

Dieser Abschnitt beinhaltet die Anfragen, die für die Evaluation des RDF-Speichermodells verwendet wurden.

A.1.1 Anfrage mit konstantem Subjekt

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.9 auf Seite 155 verwendet.

```
select * where {
       r:Review231 d:publisher ?o
}

select * where {
       r:Review231 d:publisher r:RatingSite1 ;
              d:title ?t .
}

select * where {
       r:Review231 d:publisher r:RatingSite1 ;
              v:rating2 "8" ;
              v:reviewDate ?d ;
              rdfs:type ?c .
}
```

```
select * where {
       r:Review231
              d:publisher r:RatingSite1 ;
              v:rating2 "8" ;
              s:reviewer r:Reviewer12 ;
              v:reviewFor p:Product26 ;
              v:reviewDate ?d ;
              rdfs:type ?c .
}
```

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.10 auf Seite 155 verwendet. Im Vergleich zu den vorherigen Anfragen haben die folgenden Anfragen ein größeres Ergebnis.

```
select * where {
       r:Review231 ?p ?o
}

select * where {
       r:Review231 d:publisher ?o ;
              d:title ?t .
}

select * where {
       r:Review231 d:publisher ?o ;
              v:rating2 ?t ;
              v:reviewDate ?d ;
              ?p ?c .
}
```

A.1.2 Anfrage mit variablen Subjekt

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.11 auf Seite 156 verwendet.

```
select * where {
       ?s s:reviewer r:Reviewer12
}

select * where {
       ?s d:publisher ?o ;
              s:reviewer r:Reviewer12 .
}

select * where {
       ?s d:publisher r:RatingSite1 ;
              v:rating2 "8" ;
              v:reviewDate "2007-07-03T00:00:00" ;
              rdfstype ?c .
}
```

```
select * where {
       ?s d:publisher r:RatingSite1 ;
              v:rating2 "8" ;
              s:reviewer r:Reviewer12 ;
              v:reviewFor p:Product26 ;
              v:reviewDate "2007-12-24T00:00:00" ;
              rdfs:type ?t .
}
```

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.12 auf Seite 158 verwendet. Im Vergleich zu den vorherigen Anfragen haben die folgenden Anfragen ein größeres Ergebnis.

```
select * where {
       ?s d:publisher ?p
}

select * where {
       ?s d:publisher ?o ;
              rdfs:type ?c .
}

select * where {
       ?s d:publisher ?o ;
              v:rating3 ?t ;
              v:reviewDate ?d ;
              rdfs:type ?c .
}

select * where {
       ?s d:publisher r:RatingSite1 ;
              v:rating2 "8" ;
              s:reviewer r:Reviewer12 ;
              v:reviewFor p:Product26 ;
              v:reviewDate "2007-12-24T00:00:00" ;
              rdfs:type ?t .
}
```

A.2 Anfragen mit einer Join-Operation

Dieser Abschnitt enthält die Anfragen für die Evaluation von Join-Operationen im RCS. Die gezeigten Anfragen beziehen sich auf den 10k Datensatz. Für die größeren Datensätze mussten Anfragen mit anderen Prädikaten und Objekten verwendet werden, um ähnliche Ergebnisgrößen zu erzielen. Die Struktur der Anfragen ist jedoch für alle Datensatzgrößen gleich.

A.2.1 Join zweier Sternmuster über konstantes Objekt

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.16 auf Seite 161 verwendet.

```
select * where {
       { ?p2 v:productFeature i:ProductFeature25 ;
             d:date ?d .
       }
       { ?p1 v:productFeature i:ProductFeature25 ;
             d:date ?d1 .
       }
}

select * where {
       { ?p2 v:productFeature i:ProductFeature25 ;
             d:date ?d ;
             v:productPropertyNumeric1 ?n ;
             v:productPropertyTextual2 ?t .
       }
       { ?p1 v:productFeature i:ProductFeature25 ;
             d:date ?d1 .
       }
}

select * where {
       { ?p2 v:productFeature i:ProductFeature25 ;
             d:date ?d ;
             v:productPropertyNumeric1 ?n ;
             v:productPropertyTextual2 ?t .
       }
       { ?p1 v:productFeature i:ProductFeature25 ;
             d:date ?d1 ;
             v:productPropertyNumeric1 ?n1 ;
             v:productPropertyTextual2 ?t1 .
       }
}

select * where {
       { ?p2 v:productFeature i:ProductFeature25 ;
             d:date ?d ;
             v:productPropertyNumeric1 ?n ;
             v:productPropertyTextual2 ?t ;
             v:producer ?p ;
             rdfs:label ?l .
       }
       { ?p1 v:productFeature i:ProductFeature25 ;
             d:date ?d1 .
       }
}

select * where {
       { ?p2 v:productFeature i:ProductFeature25 ;
             d:date ?d ;
             v:productPropertyNumeric1 ?n ;
```

```
            v:productPropertyTextual2 ?t ;
            v:producer ?p ;
            rdfs:label ?l .
      }
      { ?p1 v:productFeature i:ProductFeature25 ;
            d:date ?d1 ;
            v:productPropertyNumeric1 ?n1 ;
            v:productPropertyTextual2 ?t1 .
      }
}

select * where {
      { ?p2 v:productFeature i:ProductFeature25 ;
            d:date ?d ;
            v:productPropertyNumeric1 ?n ;
            v:productPropertyTextual2 ?t ;
            v:producer ?p ;
            rdfs:label ?l .
      }
      { ?p1
            v:productFeature i:ProductFeature25 ;
            d:date ?d1 ;
            v:productPropertyNumeric1 ?n1 ;
            v:productPropertyTextual2 ?t1 ;
            v:producer ?pp ;
            rdfs:label ?l1 .
      }
}
```

A.2.2 Join zwei Sternmuster über Objektvariable

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.15 auf Seite 160 verwendet.

```
select * where {
      { ?p2 v:productFeature ?f;
            d:date "2006-07-06" .
      }
      { ?p1 v:productFeature ?f;
            d:date "2004-02-14" .
      }
}

select * where {
      { ?p2 v:productFeature ?f;
            d:date "2006-07-06" ;
            v:productPropertyNumeric1 ?n ;
            v:productPropertyTextual2 ?t .
      }
      { ?p1 v:productFeature ?f;
```

```
                d:date "2004-02-14" .
        }
    }

    select * where {
        { ?p2 v:productFeature ?f;
                d:date "2006-07-06" ;
                v:productPropertyNumeric1 ?n ;
                v:productPropertyTextual2 ?t .
        }
        { ?p1 v:productFeature ?f;
                d:date "2004-02-14" ;
                v:productPropertyNumeric1 ?n1 ;
                v:productPropertyTextual2 ?t1 .
        }
    }

    select * where {
        { ?p2 v:productFeature ?f;
                d:date "2006-07-06" ;
                v:productPropertyNumeric1 ?n ;
                v:productPropertyTextual2 ?t ;
                v:producer ?p ;
                rdfs:label ?l .
        }
        { ?p1 v:productFeature ?f;
                d:date "2004-02-14" .
        }
    }

    select * where {
        { ?p2 v:productFeature ?f;
                d:date "2006-07-06" ;
                v:productPropertyNumeric1 ?n ;
                v:productPropertyTextual2 ?t ;
                v:producer ?p ;
                rdfs:label ?l .
        }
        { ?p1 v:productFeature ?f;
                d:date "2004-02-14" ;
                v:productPropertyNumeric1 ?n1 ;
                v:productPropertyTextual2 ?t1 .
        }
    }

    select * where {
        { ?p2 v:productFeature ?f;
                d:date "2006-07-06" ;
                v:productPropertyNumeric1 ?n ;
                v:productPropertyTextual2 ?t ;
```

```
            v:producer ?p ;
            rdfs:label ?l .
      }
      { ?p1 v:productFeature ?f;
            d:date "2004-02-14" ;
            v:productPropertyNumeric1 ?n1 ;
            v:productPropertyTextual2 ?t1 ;
            v:producer ?pp ;
            rdfs:label ?l1 .
      }
}
```

A.2.3 Join zwei Sternmuster über eine Subjektvariable

Die folgenden Anfragen wurden für das Erstellen von Abbildung 6.17 auf Seite 161 verwendet.

```
select * where {
      { ?r v:reviewFor ?prod ;
            d:date "2008-06-18" .
      }
      { ?prod rdfs:label ?l1 ;
            v:producer p:Producer1 .
      }
}

select * where {
      { ?r v:reviewFor ?prod ;
            d:date "2008-06-18" .
      }
      { ?prod rdfs:label ?l1 ;
            v:producer p:Producer1 ;
            d:date ?d ;
            v:productPropertyNumeric1 ?n1 .
      }
}

select * where {
      { ?r v:reviewFor ?prod ;
            s:reviewer ?rev ;
            d:title ?t ;
            d:date "2008-06-18" .
      }
      { ?prod v:producer p:Producer1 ;
            d:date ?d ;
            v:productPropertyNumeric1 ?n1 .
      }
}
```

```
select * where {
        { ?r v:reviewFor ?prod ;
              d:date "2008-06-18" .
        }
        { ?prod rdfs:label ?l1 ;
              v:producer p:Producer1 ;
              d:date ?d ;
              v:productPropertyNumeric1 ?n1 ;
              v:productPropertyNumeric2 ?n2 ;
              v:productPropertyTextual2 ?t1 .
        }
}

select * where {
        { ?r v:reviewFor ?prod ;
              s:reviewer ?rev ;
              d:title ?t ;
              d:date "2008-06-18" .
        }
        { ?prod rdfs:label ?l1 ;
              v:producer p:Producer1 ;
              d:date ?d ;
              v:productPropertyNumeric1 ?n1 ;
              v:productPropertyNumeric2 ?n2 ;
              v:productPropertyTextual2 ?t1 .
        }
}

select * where {
        { ?r v:reviewFor ?prod ;
              s:reviewer ?rev ;
              d:publisher ?pub ;
              v:reviewDate ?rd ;
              d:title ?t ;
              d:date "2008-06-18" .
        }
        { ?prod rdfs:label ?l1 ;
              v:producer p:Producer1 ;
              d:date ?d ;
              v:productPropertyNumeric1 ?n1 ;
              v:productPropertyNumeric2 ?n2 ;
              v:productPropertyTextual2 ?t1 .
        }
}
```

A.3 Evaluation graphmuster-basierte Index

Dieser Abschnitt enthält die Indexmuster und die Anfragen für die Evaluation der graphmuster-basierten Indexe.

A.3.1 Definition der Indexmuster

Die Indexmuster wurden so gewählt, dass sie kein Sternmuster komplett überlagern und drei verschiedene Indexgrößen abgedeckt werden. Für den 100k-Datensatz beinhaltet Index I_1 658, I_2 6.149 und I_3 160.829 Einträge.

A.3.1.1 Indexmuster I_1

```
select * where {
    ?x v:productFeature ?u ;
       v:productFeature i:ProductFeature1 .
}
```

A.3.1.2 Indexmuster I_2

```
select * where {
    ?x v:productFeature ?u .
    ?u d:date ?q .
}
```

A.3.1.3 Indexmuster I_3

```
select * where {
    ?x v:productFeature ?u ;
       v:productFeature ?t .
}
```

A.3.2 Definition der Anfragen

Die Anfragen wurden wie folgt konstruiert. Alle Anfragen $Q_{1,i}$ bestehen aus vier Tripelmuster, wobei drei davon ein Sternmuster bilden. Durch die Wahl eines geeigneten Objekts wird die Anzahl der Lösungen mit wachsendem i kleiner. Die Anfragen $Q_{2,i}$ und $Q_{3,i}$ wurden nach ähnlichen Kriterien konstruiert, nur dass jeweils ein weiteres Sternmuster mit zwei Tripelmustern hinzugenommen wurde.

A.3.2.1 Anfrage $Q_{1,1}$

```
select * where {
    ?x v:productFeature ?u ;
       v:productFeature ?t ;
       v:productFeature i:ProductFeature1 .
    ?u d:date ?q .
}
```

A.3.2.2 Anfrage $Q_{1,2}$

```
select * where {
       ?x v:productFeature ?u ;
              v:productFeature i:ProductFeature24 ;
              v:productFeature i:ProductFeature1 .
       ?u d:date ?q .
}
```

A.3.2.3 Anfrage $Q_{1,3}$

```
select * where {
       ?x v:productFeature ?u ;
              v:productFeature i:ProductFeature397 ;
              v:productFeature i:ProductFeature1 .
       ?u d:date ?q .
}
```

A.3.2.4 Anfrage $Q_{2,1}$

```
select * where {
       ?x v:productFeature ?u ;
              v:productFeature ?t ;
              v:productFeature i:ProductFeature1 .
       ?u d:date ?q .
       ?r v:reviewFor ?x .
       ?r v:rating3 10
}
```

A.3.2.5 Anfrage $Q_{2,2}$

```
select * where {
       ?x v:productFeature ?u ;
              v:productFeature i:ProductFeature24 ;
              v:productFeature i:ProductFeature1 .
       ?u d:date ?q .
       ?r v:reviewFor ?x .
       ?r v:rating3 10
}
```

A.3.2.6 Anfrage $Q_{2,3}$

```
select * where {
       ?x v:productFeature ?u ;
              v:productFeature i:ProductFeature397 ;
              v:productFeature i:ProductFeature1 .
       ?u d:date ?q .
       ?r v:reviewFor ?x .
       ?r v:rating3 9 .
}
```

A.3.2.7 Anfrage $Q_{3,1}$

```
select * where {
       ?x v:productFeature ?u ;
              v:productFeature ?t ;
              v:productFeature i:ProductFeature1 .
       ?u d:date ?q .
       ?r1 v:reviewFor ?x .
       ?r1 v:rating3 10 .
       ?r2 v:reviewFor ?x .
```

```
        ?r2 v:rating3 9 .
}
```

A.3.2.8 Anfrage $Q_{3,2}$

```
select * where {
        ?x v:productFeature ?u ;
                v:productFeature i:ProductFeature24 ;
                v:productFeature i:ProductFeature1 .
        ?u d:date ?q .
        ?r1 v:reviewFor ?x .
        ?r1 v:rating3 10 .
        ?r2 v:reviewFor ?x .
        ?r2 v:rating3 9 .
}
```

A.3.2.9 Anfrage $Q_{3,3}$

```
select * where {
        ?x v:productFeature ?u ;
                v:productFeature i:ProductFeature397 ;
                v:productFeature i:ProductFeature1 .
        ?u d:date ?q .
        ?r1 v:reviewFor ?x .
        ?r1 v:rating3 10 .
        ?r2 v:reviewFor ?x .
        ?r2 v:rating3 9 .
}
```

Literaturverzeichnis

[ACK+01] Alexaki, S., V. Christophides, G. Karvounarakis, D. Plexousakis und K. Tolle: *The RDFSuite: Managing Voluminous RDF Description Bases*. In: Decker, Stefan, Dieter Fensel, Amit P. Sheth und Steffen Staab (Herausgeber): *Proceedings of the 2nd International Workshop on the Semantic Web*, Seiten 1–13, 2001.

[ACZH11] Atre, Medha, Vineet Chaoji, Mohammed J. Zaki und James A. Hendler: *BitMat – Scalable Indexing and Querying of Large RDF Graphs*. Technischer Bericht, Rensselaer Polytechnic Institute, Yahoo! Labs, 2011.

[AGP10] Arenas, Marcelo, Claudio Gutierrez und Jorge Pérez: *On the Semantics of SPARQL*. In: Virgilio, Roberto de, Fausto Giunchiglia und Letizia Tanca (Herausgeber): *Semantic Web Information Management*, Seiten 281–307. Springer-Verlag, 2010.

[AMH07] Abadi, Daniel J., Adam Marcus 0002, Samuel Madden und Katherine J. Hollenbach: *Scalable Semantic Web Data Management Using Vertical Partitioning*. In: Koch, Christoph et al. (Herausgeber): *Proceedings of the 33rd International Conference on Very Large Data Bases*, Seiten 411–422. ACM Press, 2007.

[AMMH07] Abadi, Daniel J., Adam Marcus, Samuel R. Madden und Kate Hollenbach: *Using the Barton Libraries Dataset as an RDF Benchmark*. Technischer Bericht MIT-CSAIL-TR-2007-036, MIT, 2007.

[ASX01] Agrawal, Rakesh, Amit Somani und Yirong Xu: *Storage and Querying of E-Commerce Data*. In: *Proceedings of the 27th International Conference on Very Large Data Bases*, Seiten 149–158. Morgan Kaufmann Publishers Inc., 2001.

[BB07] Baolin, Liu und Hu Bo: *HPRD: A High Performance RDF Database*. In: Li, Keqiu, Chris R. Jesshope, Hai Jin und Jean-Luc Gaudiot (Herausgeber): *Proceedings of the International Conference Network and Parallel Computing*, Band 4672 der Reihe *Lecture Notes in Computer Science*, Seiten 364–374. Springer-Verlag, 2007.

[Bec04] Beckett, Dave: *RDF/XML Syntax Specification (Revised)*. http://www.w3.org/TR/rdf-syntax-grammar/, 2004. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[Bec07] Beckett, David: *Turtle – Terse RDF Triple Language*. http://www.w3.org/TeamSubmission/turtle/, 2007. Zuletzt zugegriffen: 2.3.2015.

[Ber15] *Oracle Berkeley DB*. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/, 2015. Zuletzt zugegriffen: 2.3.2015.

[BG04] Brickley, Dan und R.V. Guha: *RDF Vocabulary Description Language 1.0: RDF Schema*. http://www.w3.org/TR/rdf-schema/, 2004. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[BG06] Brass, Stefan und Christian Goldberg: *Semantic errors in SQL queries: A quite complete list*. Journal on System Software, 79(5):630–644, 2006.

[BHKN06] Beckmann, Jennifer L., Alan Halverson, Rajasekar Krishnamurthy und Jeffrey F. Naughton: *Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format*. In: *Proceedings of the 22nd International Conference on Data Engineering*, Seite 58. IEEE Computer Society, 2006.

[BKvH02] Broekstra, J., A. Kampman und F. van Harmelen: *Sesame: A generic architecture for storing and querying RDF and RDF schema*. In: Horrocks, Ian und James A. Hendler (Herausgeber): *Proceedings of the 1st International Semantic Web Conference*, Band 2342 der Reihe *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[BL06] Berners-Lee, Tim: *Notation 3*. http://www.w3.org/DesignIssues/Notation3.html, 2006. Zuletzt zugegriffen: 2.3.2015.

[BLK+09] Bizer, Christian, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak und Sebastian Hellmann: *DBpedia – A crystallization point for the Web of Data*. Journal of Web Semantics, 7(3):154–165, 2009.

[BM70] Bayer, Rudolf und Edward McCreight: *Organization and maintenance of large ordered indices*. In: *Proceedings of the Workshop on Data Description, Access and Control*, Seiten 107–141. ACM Press, 1970.

[BM04] Biron, Paul V. und Ashok Malhotra: *XML Schema Part 2: Datatypes 2nd Edition, W3C Recommendation*. http://www.w3.org/TR/xmlschema-2/, 2004. Zuletzt zugegriffen: 2.3.2015.

[Bro05] Broekstra, Jeen: *Storage, Querying and Inferencing for Semantic Web Languages*. Doktorarbeit, Vrije Universiteit, 2005. http://dare.ubvu.vu.nl/bitstream/handle/1871/8998/thesis-final.pdf. Zuletzt zugegriffen: 2.3.2015.

[BS09] Bizer, Christian und Andreas Schultz: *The Berlin SPARQL Benchmark*. Journal of Semantic Web Information Systems, 5(2):1–24, 2009.

[BS11] Bizer, Chris und Andreas Schultz: *BSBM V3 Results*. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/results/V6, 2011. Zuletzt zugegriffen: 2.3.2015.

[BSB13] *BSBM Tools*. http://sourceforge.net/projects/bsbmtools/, 2013. Zuletzt zugegriffen: 2.3.2015.

[CA+13] Chan, Immanuel, Lance Ashdown et al.: *Oracle®Database*. http://docs.oracle.com/cd/E18283_01/server.112/e16638/title.htm, 2013. Zuletzt zugegriffen: 2.3.2015.

[CBHS05] Carroll, Jeremy J., Christian Bizer, Pat Hayes und Patrick Stickler: *Named graphs, provenance and trust*. In: *Proceedings of the 14th International conference on World Wide Web*, Seiten 613–622. ACM Press, 2005.

[CDES05] Chong, Eugene Inseok, Souripriya Das, George Eadon und Jagannathan Srinivasan: *An efficient SQL-based RDF querying scheme*. In: Böhm, Klemens et al. (Herausgeber): *Proceedings of the 31st International Conference on Very Large Databases*, Seiten 1216–1227. ACM Press, 2005.

[CGM90] Chakravarthy, Upen S., John Grant und Jack Minker: *Logic-based approach to semantic query optimization*. ACM Transactions on Database Systems, 15(2):162–207, 1990.

[Cha98] Chaudhuri, Surajit: *An Overview of Query Optimization in Relational Systems*. In: *Proceedings of the 17th Symposium on Principles of Database Systems*, Seiten 34–43. ACM Press, 1998.

[Che76] Chen, Peter Pin-Shan: *The entity-relationship model – toward a unified view of data*. ACM Transactions on Database Systems, 1(1):9–36, 1976.

[CKDE03] Cruz, Isabel F., Vipul Kashyap, Stefan Decker und Rainer Eckstein (Herausgeber): *Proceedings of the 1st International Workshop on Semantic Web and Databases*, 2003.

[CLAF08] Chebotko, Artem, Shiyong Lu, Mustafa Atay und Farshad Fotouhi: *Efficient processing of RDF queries with nested optional graph patterns in an RDBMS*. International Journal on Semantic Web and Information Systems, 4(4):1–30, 2008.

[CLF09] Chebotko, Artem, Shiyong Lu und Farshad Fotouhi: *Semantics preserving SPARQL-to-SQL translation*. Data Knowledge Engineering, 68(10):973–1000, 2009.

[CLO03] Chen, Qun, Andrew Lim und Kian Win Ong: *D(k)-index: an adaptive structural summary for graph-structured data*. In: *Proceedings of the International Conference on Management of Data*, Seiten 134–144. ACM Press, 2003.

[Cod80] Codd, E. F.: *Data models in database management*. In: *Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling*, Seiten 112–114. ACM Press, 1980.

[CR02] Cho, Junghoo und Sridhar Rajagopalan: *A Fast Regular Expression Indexing Engine*. In: Agrawal, Rakesh und Klaus R. Dittrich (Herausgeber): *Proceedings of the International Conference on Data Engineering*, Seiten 419–430. IEEE Computer Society, 2002.

[CS13] Chu, Lee Jung und Sudhir Singh: *Zigzag join enablement for DB2 star schema queries*. http://www.ibm.com/developerworks/data/library/techarticle/dm-1303zigzag/, 2013. Zuletzt zugegriffen: 2.3.2015.

[Cyg05] Cyganiak, Richard: *A relational algebra for SPARQL*. Technischer Bericht HPL-2005-170, HP Laboratories Bristol, 2005.

[Dat06] Date, C.J.: *The Relational Database Dictionary: A Comprehensive Glossary of Relational Terms and Concepts, with Illustrative Examples*. O'Reilly Media, 2006.

[DPL+12] Damljanovic, Danica, Johann Petrak, Mihai Lupu, Hamish Cunningham, Mats Carlsson, Gunnar Engstrom und Bo Andersson: *Random indexing for finding similar nodes within large RDF graphs*. In: *Proceedings of the 8th International Conference on The Semantic Web*, Seiten 156–171. Springer-Verlag, 2012.

[DS09] Das, Souripriya und Jagannathan Srinivasan: *Database Technologies for RDF*. In: Tessaris, Sergio, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset und Renate A. Schmidt (Herausgeber): *Reasoning Web*, Band 5689 der Reihe *Lecture Notes in Computer Science*, Seiten 205–221. Springer-Verlag, 2009.

[ECTOO09] Elliott, Brendan, En Cheng, Chimezie Thomas-Ogbuji und Z. Meral Ozsoyoglu: *A complete translation from SPARQL into efficient SQL*. In: *IDEAS: Proceedings of the 2009 International Database Engineering & Applications Symposium*, Seiten 31–42. ACM Press, 2009.

[Esp12] Espinola, Roger Humberto Castillo: *Indexing RDF data using materialized SPARQL queries - SPARQL query processing and index selection*. Doktorarbeit, Humboldt-Universität zu Berlin, 2012. https://www.deutsche-digitale-bibliothek.de/binary/V5JW7VWUJQOIYNXKCE73UMEJPKZCBVU2/full/1.pdf. Zuletzt zugegriffen: 2.3.2015.

[FB09] Fletcher, George H.L. und Peter W. Beck: *Scalable indexing of RDF graphs for efficient join processing*. In: *Proceeding of the 18th conference on Information and knowledge management*, Seiten 1513–1516. ACM Press, 2009.

[Fra10] Franz Inc.: *AllegroGraph RDFStore™*. http://www.franz.com/agraph/allegrograph/, 2010. Zuletzt besucht am 16.3.2010.

[GPH05] Guo, Yuanbo, Zhengxiang Pan und Jeff Heflin: *LUBM: A Benchmark for OWL Knowledge Base Systems*. Web Semantics: Science, Services and Agents on the World Wide Web, 3(2-3):158–182, 2005.

[GPP13] Gearon, Paul, Alexandre Passant und Axel Polleres: *SPARQL 1.1 Update*. http://www.w3.org/TR/sparql11-update/, 2013. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[Gra93] Gray, Jim (Herausgeber): *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[GS02] Giugno, Rosalba und Dennis Shasha: *GraphGrep: A Fast and Universal Method for Querying Graphs*. Pattern Recognition, International Conference on, 2:20112, 2002.

[GSV04] Gabel, Thomas, York Sure und Johanna Voelker: *KAON - A Overview*. Technischer Bericht, Universität Karlsruhe, 2004.

[GW97] Goldman, Roy und Jennifer Widom: *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases*. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*, Seiten 436–445. Morgan Kaufmann Publishers Inc., 1997.

[GW99] Goldman, Roy und Jennifer Widom: *Approximate DataGuides*. In: *In Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Seiten 436–445, 1999.

[HB11] Heath, Tom und Christian Bizer: *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011. http://linkeddatabook.com/editions/1.0/.

[HBS08] Hertel, Alice, Jeen Broekstra und Heiner Stuckenschmidt: *Handbook on Ontologies*, Kapitel RDF Storage and Retrieval Systems, Seiten 489–508. International Handbooks on Information Systems. Springer-Verlag, 2nd Edition Auflage, 2008.

[HCL+90] Haas, Laura M., Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey und Eugene J. Shekita: *Starburst Mid-Flight: As the Dust Clears*. IEEE Transactions on Knowledge and Data Engineering, 2(1):143–160, 1990.

[HD05] Harth, Andreas und Stefan Decker: *Optimized Index Structures for Querying RDF from the Web*. In: *Proceedings of the 3rd Latin American Web Congress*. IEEE Press, 2005.

[Hee06] Heese, Ralf: *Query Graph Model for SPARQL*. In: *ER (Workshops)*, Seiten 445–454, 2006.

[HG03] Harris, Stephen und Nicholas Gibbins: *3store: Efficient Bulk RDF Storage*. In: *International Workshop on Practical and Scalable Semantic Systems*, Seiten 1–15, 2003.

[HH07] Hartig, Olaf und Ralf Heese: *The SPARQL Query Graph Model for Query Optimization*. In: *Proceedings of the 4th European conference on The Semantic Web: Research and Applications*, Seiten 564–578. Springer-Verlag, 2007.

[HL11] Huang, Hai und Chengfei Liu: *Estimating Selectivity for Joined RDF Triple Patterns*. In: *Proceedings of the 20th International Conference on Information and Knowledge Management*, Seiten 1435–1444. ACM Press, 2011.

[HLQR07] Heese, Ralf, Ulf Leser, Bastian Quilitz und Christian Rothe: *Index Support for SPARQL*. In: *Proceedings of the European Semantic Web Conference*, 2007.

[HPS14] Hayes, Patrick und Peter F. Patel-Schneider: *RDF Semantics 1.1*. http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/, 2014. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[HPSB+04] Horrocks, Ian, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof und Mike Dean: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. http://www.w3.org/Submission/SWRL/, 2004. W3C Member Submission. Zuletzt zugegriffen: 2.3.2015.

[HS05] Harris, Stephen und Nigel Shadbolt: *SPARQL query processing with conventional relational database systems*. In: Dean, M. et al. (Herausgeber): *International Workshop on Scalable Semantic Web Knowledge Base Systems*, Band 3807 der Reihe *Lecture Notes on Computer Science*, Seiten 235–244, 2005.

[HUHD07] Harth, Andreas, Jürgen Umbrich, Aidan Hogan und Stefan Decker: *YARS2: A Federated Repository for Querying Graph Structured Data from the Web*. In: Aberer, Karl et al. (Herausgeber): *ISWC/ASWC*, Band 4825 der Reihe *Lecture Notes in Computer Science*, Seiten 211–224. Springer-Verlag, 2007.

[HXF99] Han, Jiawei, Zhaohui (Alex) Xie und Yongjian Fu: *Join Index Hierarchy: An Indexing Structure for Efficient Navigation in Object-Oriented Databases*. IEEE Transactions on Knowledge and Data Engineering, 11(2):321–337, 1999.

[HZ11] Heese, Ralf und Martin Znamirowski: *Resource centered RDF data management*. In: *Proceedings of the International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2011.

[IK90] Ioannidis, Yannis E und Younkyung Kang: *Randomized algorithms for optimizing large join queries*. In: *ACM Sigmod Record*, Band 19, Seiten 312–321. ACM Press, 1990.

[Ioa96] Ioannidis, Yannis E.: *Query Optimization*. ACM Computer Survey, 28(1):121–123, 1996.

[ITL10] ITL Education Solutions Limited: *Introduction to Database Systems*. Pearson Education, 2010.

[Jen10] *Jena TDB*. http://jena.apache.org/documentation/tdb/index.html, 2010. Zuletzt zugegriffen: 2.3.2015.

[KAC+02] Karvounarakis, G., S. Alexaki, V. Christophides, D. Plexousakis und Michel Scholl: *RQL: A Declarative Query Language for RDF*. In: *Proceedings of the 1th International Conference on World Wide Web*. ACM Press, 2002.

[KC04] Klyne, Graham und Jeremy J. Carroll: *Resource Description Framework (RDF): Concepts and Abstract Syntax*. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/, 2004. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[Kim05] Kim, Sung Wan: *A Hybrid Approach to Storage Scheme for Semantic Web Metadata*. In: *Proceedings of the International Conference on Next Generation Web Services Practices*, Seite 51. IEEE Computer Society, 2005.

[KM92] Kemper, Alfons und Guido Moerkotte: *Access support relations: an indexing method for object bases*. Journal on Information Systems, 17(2):117–145, 1992.

[KSBG02] Kaushik, Raghav, Pradeep Shenoy, Philip Bohannon und Ehud Gudes: *Exploiting Local Similarity for Indexing Paths in Graph-Structured Data*. In: *Proceedings of the International Conference on Data Engineering*, Seite 0129. IEEE Computer Society, 2002.

[Ley09] Ley, Michael: *DBLP: some lessons learned*. Proceedings of the VLDB Endowment, 2(2):1493–1500, 2009.

[LG13] Loizou, Antonis und Paul T. Groth: *On the Formulation of Performant SPARQL Queries*. Computing Research Repository, abs/1304.0567, 2013.

[LKA10] Lemire, Daniel, Owen Kaser und Kamel Aouiche: *Sorting improves word-aligned bitmap indexes*. Data Knowledge Engineering, 69(1):3–28, 2010.

[LN90] Lipton, Richard J. und Jeffrey F. Naughton: *Query Size Estimation by Adaptive Sampling (Extended Abstract)*. In: *Proceedings of the 9th Symposium on Principles of Database Systems*, Seiten 40–46. ACM Press, 1990.

[Loh88] Lohman, Guy M.: *Heuristic Method for Joining Relational Database Tables*. IBM Technical Disclosure Bulletin, 30(9):8–10, 1988.

[LPL+10] Lee, Jinsoo, Minh-Duc Pham, Jihwan Lee, Wook-Shin Han, Hune Cho, Hwanjo Yu und Jeong-Hoon Lee: *Processing SPARQL queries with regular expressions in RDF databases*. In: *Proceedings of the 4th international workshop on Data and Text Mining in Biomedical Informatics*, Seiten 23–30. ACM Press, 2010.

[LR+99] Leverenz, Lefty, Diana Rehfield et al.: *Oracle8i Concepts*. http://docs.oracle.com/cd/F49540_01/DOC/server.815/a67781/c20c_joi.htm, 1999. Zuletzt zugegriffen: 2.3.2015.

[MACP02] Magkanaraki, Aimilia, Sofia Alexaki, Vassilis Christophides und Dimitris Plexousakis: *Benchmarking RDF Schemas for the Semantic Web*. In: *Proceedings of the 1st International Semantic Web Conference*, Seiten 132–146. Springer-Verlag, 2002.

[MASS08] Maduko, Angela, Kemafor Anyanwu, Amit Sheth und Paul Schliekelman: *Graph Summaries for Subgraph Frequency Estimation*. In: *Proceedings of the 5th European Semantic Web Conference*, Seiten 508–523. Springer-Verlag, 2008.

[MAYU05] Matono, Akiyoshi, Toshiyuki Amagasa, Masatoshi Yoshikawa und Shunsuke Uemura: *A path-based relational RDF database*. In: *Proceedings of the 16th Australasian conference on Database technologies*, Seiten 95–103. Australian Computer Society, Inc., 2005.

[MCS05] Min, Jun-Ki, Chin-Wan Chung und Kyuseok Shim: *An Adaptive Path Index for XML Data Using the Query Workload*. Journal on Information Systems, 30(6):467–487, 2005.

[Mic13] Microsoft®: *Advanced Query Tuning Concepts*. http://technet.microsoft.com/en-us/library/ms191426%28v=sql.105%29.aspx, 2013. Zuletzt zugegriffen: 2.3.2015.

[MS99] Milo, Tova und Dan Suciu: *Index Structures for Path Expressions*. In: *Proceedings of the 7th International Conference on Database Theory*, Seiten 277–295. Springer-Verlag, 1999.

[MSP+04] Ma, Li, Zhong Su, Yue Pan, Li Zhang und Tao Liu: *RStar: an RDF storage and query system for enterprise resource management*. In: *Proceedings of the 13th International Conference on Information and Knowledge Management*, Seiten 484–491. ACM Press, 2004.

[MSR02] Miller, Libby, Andy Seaborne und Alberto Reggiori: *Three Implementations of SquishQL, a Simple RDF Query Language*. Technischer Bericht HPL-2002-110, HP Labs, 2002.

[Mul03] Mullins, Craig: *Tuning DB2 SQL Access Paths*. http://www.ibm.com/developerworks/data/library/techarticle/0301mullins/0301mullins.html, 2003. Zuletzt zugegriffen: 2.3.2015.

[MWL+08] Ma, Li, Chen Wang, Jing Lu, Feng Cao, Yue Pan und Yong Yu: *Effective and efficient semantic web data management over DB2*. In: *Proceedings International Conference on Management of data*, Seiten 1183–1194. ACM Press, 2008.

[NW10] Neumann, Thomas und Gerhard Weikum: *The RDF-3X engine for scalable management of RDF data*. The VLDB Journal, 19(1):91–113, 2010.

[OBW06] Oldakowski, Radoslaw, Christian Bizer und Daniel Westphal: *RAP: RDF API for PHP*. In: Auer, Sören, Chris Bizer und Libby Miller (Herausgeber): *Proceedings of the Workshop on Scripting for the Semantic Web*, Band 135, 2006.

[Ope10] OpenLink Software: *Virtuoso*. http://virtuoso.openlinksw.com/, 2010. Zuletzt zugegriffen: 2.3.2015.

[Ora14] Oracle: *Oracle Spatial and Graph: Advanced Data Management*, 2014.

[PAG09] Pérez, Jorge, Marcelo Arenas und Claudio Gutierrez: *Semantics and complexity of SPARQL*. ACM Transactions on Database Systems, 34(3):16:1–16:45, 2009.

[PI97] Poosala, Viswanath und Yannis E. Ioannidis: *Selectivity Estimation Without the Attribute Value Independence Assumption*. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*, Seiten 486–495. Morgan Kaufmann Publishers Inc., 1997.

[Pos13] *PostgreSQL*. http://www.postgresql.org, 2013. Zuletzt zugegriffen: 2.3.2015.

[PPD+13] Potocki, Alexander, Anton Polukhin, Grigory Drobyazko, Daniel Hladky, Victor P. Klintsov und Jörg Unbehauen: *OntoQuad: Native High-Speed RDF DBMS for Semantic Web*. In: Klinov, Pavel und Dmitry Mouromtsev (Herausgeber): *Knowledge Engineering and the Semantic Web*, Band 394 der Reihe *Communications in Computer and Information Science*, Seiten 117–131. Springer-Verlag, 2013.

[PS08] Prud'hommeaux, Eric und Andy Seaborne: *SPARQL Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/, 2008. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[PS13] Prud'hommeaux, Eric und Andy Seaborne: *SPARQL 1.1 Query Language for RDF*. http://www.w3.org/TR/rdf-sparql-query/, 2013. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[Rot12] Rothe, C.: *Indexierung von RDF-Daten für SPARQL-Anfragen: Auswahl einer optimalen Indexmenge für eine gegebene Anfrage*. AV Akademikerverlag, 2012.

[RSSS94] Ramakrishnan, Raghu, Divesh Srivastava, S. Sudarshan und Praveen Seshadri: *The CORAL deductive system*. The VLDB Journal, 3(2):161–210, 1994.

[SAC+79] Selinger, Patricia G., Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie und Thomas G. Price: *Access Path Selection in a Relational Database Management System*. In: Bernstein, Philip A. (Herausgeber): *Proceedings of the International Conference on Management of Data*, Seiten 23–34. ACM Press, 1979.

[Sch10] Schmidt, Michael: *Foundations of SPARQL query optimization*. Doktorarbeit, Universität Freiburg, 2010. http://www2.informatik.uni-freiburg.de/~mschmidt/docs/diss_final01122010.pdf. Zuletzt zugegriffen: 2.3.2015.

[SHLP09] Schmidt, Michael, Thomas Hornung, Georg Lausen und Christoph Pinkel: *SP^2Bench: A SPARQL Performance Benchmark*. In: *ICDE*, Seiten 222–233. IEEE, 2009.

[SMK97] Steinbrunn, Michael, Guido Moerkotte und Alfons Kemper: *Heuristic and randomized optimization for the join ordering problem*. The VLDB Journal, 6(3):191–208, 1997.

[SSB+08] Stocker, Markus, Andy Seaborne, Abraham Bernstein, Christoph Kiefer und Dave Reynolds: *SPARQL basic graph pattern optimization using selectivity estimation*. In: *Proceedings of the 17th International Conference on World Wide Web*, Seiten 595–604. ACM Press, 2008.

[SVHB04] Stuckenschmidt, Heiner, Richard Vdovjak, Geert-Jan Houben und Jeen Broekstra: *Index structures and algorithms for querying distributed RDF repositories*. In: *Proceedings of the 13th International Conference on World Wide Web*, Seiten 631–639. ACM Press, 2004.

[TCK05] Theoharis, Yannis, Vassilis Christophides und Gregory Karvounarakis: *Benchmarking Database Representations of RDF/S Stores*. In: *International Semantic Web Conference*, Lecture Notes in Computer Science, Seiten 685–701. Springer-Verlag, 2005.

[TGC12] Theoharis, Yannis, George Georgakopoulos und Vassilis Christophides: *PoweRGen: A power-law based generator of RDFS schemas*. Journal on Information Systems, 37(4):306–319, 2012.

[TSF+12] Tsialiamanis, Petros, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides und Peter Boncz: *Heuristics-based query optimisation for SPARQL*. In: *Proceedings of the 15th International Conference on Extending Database Technology*, Seiten 324–335. ACM Press, 2012.

[TTKC08] Theoharis, Yannis, Yannis Tzitzikas, Dimitris Kotzinos und Vassilis Christophides: *On Graph Features of Semantic Web Schemas*. IEEE Transactions on Knowledge and Data Engineering, 20(5):692–702, 2008.

[Ull80] Ullman, Jeffrey D.: *Principles of database systems*. Computer Science Press, 1980.

[UPS07] Udrea, Octavian, Andrea Pugliese und V. S. Subrahmanian: *GRIN: a graph based RDF index*. In: *Proceedings of the 22nd national conference on Artificial intelligence*, Band 2, Seiten 1465–1470. AAAI Press, 2007.

[V+10] Vidal, Maria-Esther et al.: *Efficiently Joining Group Patterns in SPARQL Queries*. In: Aroyo, Lora et al. (Herausgeber): *7th Extended Semantic Web Conference*, Band 6088 der Reihe *Lecture Notes in Computer Science*, Seiten 228–242. Springer-Verlag, 2010.

[W3C12] W3C OWL Working Group: *OWL 2 Web Ontology Language Document Overview (2nd Edition)*. http://www.w3.org/TR/owl2-overview/, 2012. W3C Recommendation. Zuletzt zugegriffen: 2.3.2015.

[WAH07] *WAH Compressed BitSet for Java*. http://code.google.com/p/compressedbitset/, 2007. Zuletzt zugegriffen: 2.3.2015.

[WGA05] Wood, David, Paul Gearon und Tom Adams: *Kowari: a platform for semantic web storage and analysis*. In: *Proceedings of XTech*, 2005.

[WGQH05] Wang, Sui-yu, Yuanbo Guo, Abir Qasem und Jeff Heflin: *Rapid benchmarking for semantic web knowledge base systems*. In: *Proceedings of the 4th International Semantic Web Conference*, Seiten 745–757, 2005.

[WKB08] Weiss, Cathrin, Panagiotis Karras und Abraham Bernstein: *Hexastore: sextuple indexing for semantic web data management*. Proceedings of the VLDB Endowment, 1(1):1008–1019, 2008.

[WLHW09] Wu, Gang, Juanzi Li, Jianqiang Hu und Kehong Wang: *System Pi: A Native RDF Repository Based on the Hypergraph Representation for RDF Data Model*. Journal of Computer Science and Technology, 24(4):652–664, 2009.

[WLS03] Weithöner, Timo, Thorsten Liebig und Günther Specht: *Storing and Querying Ontologies in Logic Databases*. In: Cruz, Isabel F. et al. [CKDE03].

[WOS06] Wu, Kesheng, Ekow J. Otoo und Arie Shoshani: *Optimizing bitmap indices with efficient compression*. ACM Transactions on Database Systems, 31(1):1–38, 2006.

[WSKR03] Wilkinson, Kevin, Craig Sayers, Harumi Kuno und Dave Reynolds: *Efficient RDF Storage and Retrieval in Jena2*. In: Cruz, Isabel F. et al. [CKDE03].

[ZzC+13] Zou, Lei, M.Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang und Dongyan Zhao: *gStore: a graph-based SPARQL query engine*. VLDB Journal, (8):1–26, 2013.

Erklärung

Hiermit erkläre ich,

- dass ich die vorliegende Arbeit mit dem Titel „Resource Centered Store" selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt und sie an keiner anderen Universität eingereicht habe,
- dass ich keinen Doktorgrad im Fach Informatik besitze,
- und dass mir die aktuelle Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Ralf Heese